

```
import React, { Component } from
'react';

import './App.css';
import Card from './Card';

export default class App extends
Component {
  render() {
    return (
      <div>
        <h1>Card Game</h1>
        <Card />
      </div>
    );
  }
}
```



Programming
Libraries,
Components,
Frontend,
JavaScript,
JSX, Props,
Event-binding

React

The Comprehensive Guide

Sebastian Springer

 Rheinwerk
Computing

Rheinwerk Computing

The Rheinwerk Computing series from Rheinwerk Publishing offers new and established professionals comprehensive guidance to enrich their skillsets and enhance their career prospects. Our publications are written by leading experts in the fields of programming, administration, security, analytics, and more. Each book is detailed and hands-on to help readers develop essential, practical skills that they can apply to their daily work. For further information, please visit our website: www.rheinwerk-computing.com.

Philip Ackermann

JavaScript: The Comprehensive Guide

2022, approx. 1292 pp, paperback and e-book

www.rheinwerk-computing.com/5554

Sebastian Springer

Node.js: The Comprehensive Guide

2022, 834 pages, paperback and e-book

www.rheinwerk-computing.com/5556

Jürgen Wolf

HTML and CSS: The Comprehensive Guide

2023, 814 pages, paperback and e-book

www.rheinwerk-computing.com/5695

Philip Ackermann

Full Stack Web Development: The Comprehensive Guide

2023, 740 pages, paperback and e-book

www.rheinwerk-computing.com/5704

Johannes Ernesti, Peter Kaiser

Python 3: The Comprehensive Guide

2022, 1036 pages, paperback and e-book

www.rheinwerk-computing.com/5566

Sebastian Springer

React

The Comprehensive Guide

Imprint

This e-book is a publication many contributed to, specifically:

Editor Rachel Gibson

Acquisitions Editor Hareem Shafi

German Edition Editor Patricia Schiewald

Translation Winema Language Services, Inc.

Copyeditor Melinda Rankin

Cover Design Graham Geary

Photo Credit Shutterstock: 2265417157/© Oleksandr Antonov;

iStockphoto: 1385385993/© D3Damon

Layout Design Vera Brauner

Production E-Book Hannah Lane

Typesetting E-Book Ill-satz, Germany

We hope that you liked this e-book. Please share your feedback with us and read the [Service Pages](#) to find out how to contact us.

The Library of Congress has cataloged the printed edition Cataloging-in-Publication Number as follows: 2023945633

ISBN 978-1-4932-2440-1 (print)

ISBN 978-1-4932-2441-8 (e-book)

ISBN 978-1-4932-2442-5 (print and e-book)

© 2024 by Rheinwerk Publishing, Inc., Boston (MA)

1st edition 2024

2nd German edition published 2023 by Rheinwerk Verlag, Bonn, Germany

Contents

Foreword	21
Preface	23
1 Getting Started with React	27
1.1 What Is React?	27
1.1.1 Single-Page Applications	28
1.1.2 The Story of React	28
1.2 Why React?	32
1.3 The Most Important Terms and Concepts of the React World	34
1.3.1 Components and Elements	34
1.3.2 Data Flow	36
1.3.3 The Renderer	37
1.3.4 The Reconciler	37
1.4 A Look into the React Universe	40
1.4.1 State Management	40
1.4.2 The Router	40
1.4.3 Material UI	40
1.4.4 Jest	41
1.5 Thinking in React	41
1.5.1 Decomposing the UI into a Component Hierarchy	41
1.5.2 Implementing a Static Version in React	42
1.5.3 Defining the Minimum UI State	42
1.5.4 Defining the Location of the State	42
1.5.5 Modeling the Inverse Data Flow	42
1.6 Code Examples	42
1.7 Summary	43
2 The First Steps in the Development Process	45
2.1 Quick Start	45
2.1.1 Initialization	45
2.1.2 TypeScript Support	46

- 2.2 Playgrounds for React** 46
 - 2.2.1 CodePen: A Playground for Web Development 46
 - 2.2.2 A React Project on CodePen 47
- 2.3 Local Development** 49
- 2.4 Getting Started with Developing in React** 52
 - 2.4.1 Requirements 53
 - 2.4.2 Installing Create React App 54
 - 2.4.3 Alternatives to Create React App 61
 - 2.4.4 React Scripts 62
 - 2.4.5 Server Communication in Development Mode 64
 - 2.4.6 Encrypted Communication during Development 66
- 2.5 The Structure of the Application** 66
- 2.6 Troubleshooting in a React Application** 68
- 2.7 Building the Application** 71
- 2.8 Summary** 71

3 Basic Principles of React 73

- 3.1 Preparation** 73
 - 3.1.1 Tidying Up the Application 74
- 3.2 Getting Started with the Application** 74
 - 3.2.1 The index.jsx File: Rendering the Application 74
 - 3.2.2 The App.jsx File: The Root Component 77
- 3.3 Function Components** 78
 - 3.3.1 One Component per File 80
- 3.4 JSX: Defining Structures in React** 84
 - 3.4.1 Expressions in JSX 86
 - 3.4.2 Iterations: Loops in Components 89
 - 3.4.3 Conditions in JSX 91
- 3.5 Props: Information Flow in an Application** 95
 - 3.5.1 Props and Child Components 95
 - 3.5.2 Type Safety with PropTypes 97
- 3.6 Local State** 100
- 3.7 Event Binding: Responding to User Interactions** 102
 - 3.7.1 Responding to Events 102
 - 3.7.2 Using Event Objects 107

3.8	Immutability	110
3.8.1	Immer in a React Application	111
3.9	Summary	113
4	A Look Behind the Scenes: Further Topics	115
4.1	The Lifecycle of a Component	115
4.2	The Lifecycle of a Function Component with the Effect Hook	116
4.2.1	Mount: The Mounting of a Component	116
4.2.2	Update: Updating the Component	119
4.2.3	Unmount: Tidying Up at the End of the Lifecycle	123
4.3	Server Communication	126
4.3.1	Server Implementation	126
4.3.2	Server Communication via the Fetch API	128
4.3.3	Things to Know about Server Communication	132
4.3.4	Server Communication with Axios	135
4.4	Container Components	137
4.4.1	Swapping Out Logic to a Container Component	137
4.4.2	Integrating the Container Component	140
4.4.3	Implementing the Presentational Component	140
4.5	Higher-Order Components	142
4.5.1	A Simple Higher-Order Component	142
4.5.2	Integrating a Higher-Order Component in the BooksList Component	144
4.5.3	Integrating the Higher-Order Component	145
4.6	Render Props	146
4.6.1	Alternative Names for Render Props	147
4.6.2	Integrating the Render Props into the Application	149
4.7	Context	150
4.7.1	The Context API	151
4.7.2	Using the Context API in the Sample Application	154
4.8	Fragments	158
4.9	Summary	159

5 Class Components 161

5.1	Class Components in React	161
5.2	Basic Structure of a Class Component	162
5.3	Props in a Class Component	163
5.3.1	Defining Prop Structures Using PropTypes	164
5.3.2	Default Values for Props	165
5.4	State: The State of the Class Component	166
5.4.1	Initializing the State via the State Property of the Class	166
5.4.2	Initializing the State in the Constructor	167
5.5	The Component Lifecycle	169
5.5.1	Constructor	171
5.5.2	“getDerivedStateFromProps”	171
5.5.3	“render”	172
5.5.4	“componentDidMount”	173
5.5.5	“shouldComponentUpdate”	174
5.5.6	“getSnapshotBeforeUpdate”	175
5.5.7	“componentDidUpdate”	175
5.5.8	“componentWillUnmount”	177
5.5.9	Unsafe Hooks	178
5.6	Error Boundaries	179
5.6.1	Logging Errors Using “componentDidCatch”	180
5.6.2	Alternative Representation in Case of an Error with “getDerivedStateFromError”	182
5.7	Using the Context API in a Class Component	183
5.8	Differences between Function and Class Components	185
5.8.1	State	185
5.8.2	Lifecycle	186
5.9	Summary	186

6 The Hooks API of React 187

6.1	A First Overview	188
6.1.1	The Three Basic Hooks	188
6.1.2	Other Components of the Hooks API	188
6.2	“useReducer”: The Reducer Hook	190
6.2.1	The Reducer Function	192

6.2.2	Actions and Dispatching	192
6.2.3	Asynchronicity in the Reducer Hook	193
6.3	“useCallback”: Memoizing Functions	197
6.4	“useMemo”: Memoizing Objects	198
6.5	“useRef”: References and Immutable Values	200
6.5.1	Form Handling Using the Ref Hook	200
6.5.2	Caching Values Using the Ref Hook	201
6.6	“useImperativeHandle”: Controlling Forward Refs	202
6.6.1	Forward Refs	202
6.6.2	The Imperative Handle Hook	204
6.7	“useLayoutEffect”: The Synchronous Alternative to “useEffect”	205
6.8	“useDebugValue”: Debugging Information in React Developer Tools	206
6.9	“useDeferredValue”: Performing Updates According to Priority	207
6.10	“useTransition”: Lowering the Priority of Operations	210
6.11	“useId”: Creating Unique Identifiers	212
6.12	Library Hooks	213
6.12.1	“useSyncExternalStore”	213
6.12.2	“useInsertionEffect”	213
6.13	Custom Hooks	214
6.14	Rules of Hooks: Things to Consider	215
6.14.1	Rule #1: Execute Hooks Only at the Top Level	216
6.14.2	Rule #2: Hooks May Only Be Used in Function Components or Custom Hooks	216
6.15	Changing over to Hooks	217
6.16	Summary	218

7 Type Safety in React Applications with TypeScript 219

7.1	What Is the Benefit of a Type System?	219
7.2	The Different Type Systems	220
7.3	Type Safety in a React Application with Flow	220
7.3.1	Integration into a React Application	221
7.3.2	The Major Features of Flow	224
7.3.3	Flow in React Components	224

- 7.4 Using TypeScript in a React Application** 225
 - 7.4.1 Integrating TypeScript in a React Application 226
 - 7.4.2 Configuration of TypeScript 229
 - 7.4.3 The Major Features of TypeScript 230
 - 7.4.4 Type Definitions: Information about Third-Party Software 230
- 7.5 TypeScript and React** 231
 - 7.5.1 Adding TypeScript to an Existing Application 231
 - 7.5.2 Basic Features 231
 - 7.5.3 Function Components 235
 - 7.5.4 Context 238
 - 7.5.5 Class Components 241
- 7.6 Summary** 242

8 Styling React Components 245

- 8.1 CSS Import** 245
 - 8.1.1 Advantages and Disadvantages of CSS Import 246
 - 8.1.2 Dealing with Class Names 248
 - 8.1.3 Improved Handling of Class Names via the “classnames” Library 250
 - 8.1.4 Using Sass as CSS Preprocessor 252
- 8.2 Inline Styling** 253
- 8.3 CSS Modules** 255
- 8.4 CSS in JavaScript Using Emotion** 257
 - 8.4.1 Installing Emotion 258
 - 8.4.2 Using the “css” Prop 258
 - 8.4.3 The Styled Approach of Emotion 260
 - 8.4.4 Pseudoselectors in Styled Components 262
 - 8.4.5 Dynamic Styling 263
 - 8.4.6 Other Features of Styled Components 265
- 8.5 Tailwind** 266
- 8.6 Summary** 268

9 Securing a React Application through Testing 271

- 9.1 Getting Started with Jest** 272
 - 9.1.1 Installation and Execution 273
 - 9.1.2 Organization of the Tests 274

9.1.3	Jest: The Basic Principles	274
9.1.4	Structure of a Test: Triple A	276
9.1.5	The Matchers of Jest	277
9.1.6	Grouping Tests: Test Suites	279
9.1.7	Setup and Teardown Routines	279
9.1.8	Skipping Tests and Running Them Exclusively	281
9.1.9	Handling Exceptions	283
9.1.10	Testing Asynchronous Operations	284
9.2	Testing Helper Functions	287
9.3	Snapshot Testing	289
9.4	Testing Components	294
9.4.1	Testing the “BooksListItem” Component	294
9.4.2	Testing the Interaction	297
9.5	Dealing with Server Dependencies	299
9.5.1	Simulating Errors during Communication	301
9.6	Summary	303

10 Forms in React 305

10.1	Uncontrolled Components	305
10.1.1	Handling References in React	306
10.2	Controlled Components	318
10.3	File Uploads	328
10.4	Form Validation Using React Hook Form	334
10.4.1	Form Validation Using React Hook Form	337
10.4.2	Form Validation Using a Schema	339
10.4.3	Styling the Form	342
10.4.4	Testing the Form Validation Automatically	343
10.5	Summary	345

11 Component Libraries in a React Application 347

11.1	Installing and Integrating Material UI	347
11.2	List Display with the “Table” Component	348
11.2.1	Filtering the List in the Table	352
11.2.2	Sorting the Table	355

11.3	Grids and Breakpoints	358
11.4	Icons	361
11.5	Deleting Data Records	363
11.5.1	Preparing a Delete Operation	364
11.5.2	Implementing a Confirmation Dialog	365
11.5.3	Deleting Data Records	367
11.6	Creating New Data Records	369
11.6.1	Preparing the Creation of Data Records	369
11.6.2	Implementation of the “Form” Component	370
11.6.3	Integration of the Form Dialog	373
11.7	Editing Data Records	376
11.8	Summary	381

12	Navigating Within an Application: The Router	383
-----------	---	-----

12.1	Installation and Integration	383
12.2	Navigating in the Application	385
12.2.1	The Best Route Is Always Activated	387
12.2.2	A Navigation Bar for the Application	387
12.2.3	Integrating the Navigation Bar	389
12.3	“NotFound” Component	390
12.4	Testing the Routing	392
12.5	Conditional Redirects	396
12.6	Dynamic Routes	400
12.6.1	Defining Subroutes	400
12.7	Summary	406

13	Creating Custom React Libraries	407
-----------	--	-----

13.1	Creating a Custom Component Library	407
13.1.1	Initializing the Library	408
13.1.2	The Structure of the Library	411
13.1.3	Hooks in the Library	413
13.1.4	Building the Library	414

13.2 Integrating the Library	415
13.2.1 Regular Installation of the Package	418
13.3 Testing the Library	419
13.3.1 Preparing the Testing Environment	419
13.3.2 Unit Test for the Library Component	419
13.3.3 Unit Test for the Custom Hook of the Library	420
13.4 Storybook	422
13.4.1 Installing and Configuring Storybook	422
13.4.2 Button Story in Storybook	423
13.5 Summary	425

14 Central State Management Using Redux 427

14.1 The Flux Architecture	427
14.1.1 The Central Data Store: The Store	428
14.1.2 Displaying the Data in the Views	428
14.1.3 Actions: The Description of Changes	429
14.1.4 The Dispatcher: The Interface between Actions and the Store	430
14.2 Installing Redux	431
14.3 Configuring the Central Store	431
14.3.1 Debugging Using the Redux Dev Tools	433
14.4 Handling Changes to the Store Using Reducers	435
14.4.1 The Books Slice	435
14.4.2 Integration of “BooksSlice”	437
14.5 Linking Components and the Store	438
14.5.1 Displaying the Data from the Store	439
14.5.2 Selectors	440
14.5.3 Implementing Selectors Using Reselect	443
14.6 Describing Changes with Actions	444
14.7 Creating and Editing Data Records	448
14.8 Summary	453

15 Handling Asynchronicity and Side Effects in Redux 455

15.1 Middleware in Redux	455
---------------------------------	-----

15.2	Redux with Redux Thunk	457
15.2.1	Manual Integration of Redux Thunk	457
15.2.2	Reading Data from the Server	458
15.2.3	Deleting Data Records	465
15.2.4	Creating and Modifying Data Records	470
15.3	Generators: Redux Saga	473
15.3.1	Installation and Integration of Redux Saga	475
15.3.2	Loading Data from the Server	476
15.3.3	Deleting Existing Data	482
15.3.4	Creating and Modifying Data Records Using Redux Saga	487
15.4	State Management Using RxJS: Redux Observable	491
15.4.1	Installing and integrating Redux Observable	492
15.4.2	Read Access to the Server Using Redux Observable	493
15.4.3	Deleting Using Redux Observable	495
15.4.4	Creating and Editing Data Records Using Redux Observable	496
15.5	JSON Web Token for Authentication	498
15.6	Summary	507

16 Server Communication Using GraphQL and the Apollo Client

16.1	Introduction to GraphQL	509
16.1.1	The Characteristics of GraphQL	509
16.1.2	The Disadvantages of GraphQL	510
16.1.3	The Principles of GraphQL	511
16.2	Apollo: A GraphQL Client for React	514
16.2.1	Installation and Integration into the Application	515
16.2.2	Read Access to the GraphQL Server	516
16.2.3	States of a Request	518
16.2.4	Type Support in the Apollo Client	520
16.2.5	Deleting Data Records	522
16.3	Apollo Client Devtools	526
16.4	Local State Management Using Apollo	527
16.4.1	Initializing the Local State	527
16.4.2	Using the Local State	528
16.5	Authentication	532
16.6	Summary	534

17 Internationalization 535

17.1 Using react-i18next	535
17.1.1 Loading Language Files from the Backend	540
17.1.2 Using the Language of the Browser	541
17.1.3 Extending the Navigation with Language Switching	542
17.2 Using Placeholders	544
17.3 Formatting Values	546
17.3.1 Formatting Numbers and Currencies	547
17.3.2 Formatting Date Values	549
17.4 Singular and Plural	551
17.5 Summary	553

18 Universal React Apps with Server-Side Rendering 555

18.1 How Does Server-Side Rendering Work?	556
18.2 Implementing Server-Side Rendering	557
18.2.1 Initializing and Configuring the Server Application	557
18.2.2 Implementing the Client-Side Application	563
18.2.3 Dynamics in Server-Side Rendering	565
18.3 Server-Side Rendering Using Next.js	569
18.3.1 Initializing a Next.js Application	569
18.3.2 Implementing the Page Component	569
18.3.3 Implementing the Server Side	571
18.3.4 API Routes in Next.js	573
18.4 Summary	574

19 Performance 577

19.1 The Callback Hook	577
19.2 Pure Components	580
19.3 “React.memo”	581
19.4 “React.lazy”: “Suspense” for Code Splitting	584
19.4.1 Lazy Loading in an Application	584
19.4.2 Lazy Loading with React Router	588

- 19.5 Suspense for Data Fetching** 591
 - 19.5.1 Installing and Using React Query 592
 - 19.5.2 React Query and Suspense 596
 - 19.5.3 Concurrency Patterns 598
- 19.6 Virtual Tables** 599
- 19.7 Summary** 603

20 Progressive Web Apps 605

- 20.1 Features of a Progressive Web App** 605
- 20.2 Initializing the Application** 606
- 20.3 Installability** 606
 - 20.3.1 Secure Delivery of an Application 607
 - 20.3.2 The Web App Manifest 609
 - 20.3.3 Service Worker in the React Application 612
 - 20.3.4 Installing the Application 613
 - 20.3.5 Asking the Users 614
- 20.4 Offline Capability** 618
 - 20.4.1 Integrating Workbox 619
 - 20.4.2 Handling Dynamic Data 623
- 20.5 Tools for Development** 628
- 20.6 Summary** 629

21 Native Apps with React Native 631

- 21.1 The Structure of React Native** 631
- 21.2 Installing React Native** 631
 - 21.2.1 Project Structure 632
 - 21.2.2 Starting the Application 633
- 21.3 Displaying an Overview List** 636
 - 21.3.1 Static List View 636
 - 21.3.2 Styling in React Native 639
 - 21.3.3 Search Field for the “List” Component 644
 - 21.3.4 Server Communication 646
- 21.4 Debugging in the Simulated React Native Environment** 648

21.5 Editing Data Records	650
21.5.1 Implementing the “Form” Component	652
21.6 Publishing	657
21.7 Summary	658
The Author	661
Index	663
Service Pages	I
Legal Notes	II

Foreword

Web development is advancing continuously, which is both a curse and a blessing for web developers. Hardly a week goes by without another new library or framework being hyped in the community and celebrated as the supposed savior. Developers with experience know that such hype should be treated with caution. But don't misunderstand me: there is often something to hype, but as with everything, you should use your common sense and not jump onto every new bandwagon without being asked. One bandwagon that has been running for a long time and can be jumped onto without hesitation is React, a library developed by Facebook.

The year 2023 marks the 10th anniversary of React, which means it's already a prehistoric rock if you consider the fast pace of software. It's both nice and reassuring to know both that React is now firmly established and that developers are regularly adding new aspects and features. In addition, React is very well thought-out in terms of its underlying concepts, which in turn makes it extraordinarily performant and—compared to other large frameworks—lets it move in an extremely attractive cosmos—be it the alternative package manager Yarn, which is also developed by Meta, the company behind Facebook, and which has already inspired a couple of features for the official Node.js package manager, npm; or the Jest testing tool, which is particularly suitable for testing React applications thanks to snapshot testing. GraphQL, the more flexible alternative to REST, also comes from Meta and integrates seamlessly with React applications. The total of all this is why React remains my personal preference over other comparable libraries and frameworks.

For newcomers, there are of course a few hurdles to overcome with React. Concepts like Redux, JSX, and Hooks need to be understood before they can be used effectively. And this is exactly where this completely revised second edition by Sebastian Springer will help. You'll learn how to best set up, organize, and plan React applications, how to structure components, and best practices to follow. Particularly noteworthy is that Sebastian manages to explain topics in individual, self-contained chapters, which nevertheless follow a common thread and build on each other in a didactically outstanding manner.

As a reviewer, I had the pleasure of reading both German editions (or the respective manuscripts) in advance and have to say: in this English edition, Sebastian achieves perfection. Not only are newer features presented and described so that you are brought up to date with React, but Sebastian has also taken the trouble to revise the entire book, which has an extremely positive effect on your reading experience. My respect, Sebastian! And to you, dear readers, enjoy the book and learning React.

Philip Ackermann

Chief Technology Officer at Cedalo GmbH

Preface

React has established itself as a force to be reckoned with in the frontend world over the past few years, and it's hard to imagine the world without it as one of the big three solutions alongside Angular and Vue. To me, React stands for a lightweight and flexible way of development that is nevertheless highly professional. It gives you a high degree of freedom when building and designing your application. This is both a curse and a blessing. Especially for beginners, that's the point where it gets difficult: Where should I start? How do I structure my application? How do I solve specific problems? What libraries and tools do I need to develop my application? These are just a few questions to ask yourself as you begin your journey with React, and that's where this book comes in. Together we'll implement a complete application that covers numerous problems from everyday practice. In the process, you'll learn about not only React itself, but also parts of the ecosystem around the library. Whether you're just getting started with React or already have experience with it, I'd like to invite you to use this book as an opportunity to actively work with React. Use the code samples, extend them, or build your own applications. Try to implement different requirements and be inspired by the code examples and approaches for your own solutions. There is hardly a better strategy for delving deeper into a topic than using the technology or tool yourself, even making a mistake once in a while and learning from it.

To work with this book, you should have a solid basic knowledge of HTML, CSS, and JavaScript. If you are unsure at one point or another, or are wondering what a particular language element does exactly, I recommend the Mozilla Developer Network at <https://developer.mozilla.org>. This is a comprehensive up-to-date reference for all web technologies. If you prefer to work with books, I can recommend *JavaScript: The Comprehensive Guide* by Philip Ackermann. Otherwise, I recommend that you should be curious and try things out. Ask yourself: What happens if I do this or that at this specific point? Try it out, open your browser's developer tools, and see the effects of your experiment. The advantage of frontend web development is that you can't break anything else in your application except the frontend, and you can also keep that to a minimum by using a version control system like Git as you can always revert back to a working state. In addition to these experiments, you should also try to develop the sample application independently or build your own application.

This book is designed both to get you started with React and as a reference for everyday use. You can either follow the examples by writing the source code yourself or download the code and customize it as you like. All that really matters to me is that you start using React yourself, get to know the library and its capabilities, and have a lot of fun doing it.

One of the most common questions related to JavaScript libraries and frameworks is which one is the best. Of course, React is a good choice when it comes to implementing a web frontend. However, other solutions like Angular, Vue, Svelte, and many more are not worse at all. Try to make up your own mind at this point and give the different approaches a chance. To me, React has proven itself in practice in both small and large projects.

This second edition reflects the evolution of React and the ecosystem around the library. In addition, a great deal of feedback from readers of the first edition has been incorporated. Thus, the book no longer relies on one continuous and, toward the end of the book, very extensive example, but on smaller sample applications. Although all of them deal with the topic of book management, they do not build on each other in a linear fashion.

Before I briefly introduce you to the structure of this book, I would like to give you a hint along the way: the world of JavaScript frameworks and libraries is extremely fast-paced, and so you will find that the version numbers and, to some extent, the features of the libraries presented here will change. That's why in this book I also provide you with background information on concepts and architecture patterns that will allow you to adapt to changes and get up to speed very quickly on new features and libraries.

Structure of the Book

The book consists of two parts and is divided into a total of 21 chapters. The first part of the book deals with React itself, while the second part highlights the ecosystem of the library with various problems from everyday project work.

The first part starts with an introduction to React, explaining the basic principles and the most important terms and concepts (**Chapter 1**), followed by an explanation of how to install and use React (**Chapter 2**). You will then learn how to use React components in **Chapter 3** and **Chapter 4**. This involves building the component tree of your application, the flow of data in this tree, and the way the lifecycle of a component works. More advanced concepts such as higher-order components and render props complement these basics. In **Chapter 5**, I'll introduce the now obsolete class components. The main reason for this is that you will always encounter this type of component in existing applications. **Chapter 6** is dedicated to the Hooks API, an extension to React that significantly affects the way an application is built. In **Chapter 7**, you will learn about TypeScript, a tool that supports you in implementation and provides additional security in the development process.

React supports different approaches when it comes to styling components. **Chapter 8** introduces you to some of these approaches and shows you how you can integrate them into your application. Another tool related to the quality assurance of an application is covered in **Chapter 9**, which deals with formulating unit tests. **Chapter 10**

concludes the first part of the book with a look at forms. Using forms, you can give your users the opportunity to actively interact with an application and produce data.

The second part of the book begins with a chapter about the integration of external component libraries (**Chapter 11**). Using Material UI as an example, you'll see how you can integrate existing components into your application. In **Chapter 12**, you'll learn how to use the React router to navigate within your single-page application and use it to render different subtrees of your application. **Chapter 13** deals with custom component libraries. This is an important topic when it comes to reusing components across different applications. **Chapter 14** and **Chapter 15** deal with centralized state management in an application with the Redux library and with handling side effects with various asynchronous middleware implementations such as Redux-Thunk.

In **Chapter 16**, you'll learn how to address a GraphQL interface in your React application. This query language provides a flexible alternative to the traditional RESTful interfaces commonly used in web development. **Chapter 17** integrates `react-i18next`, another external library, into the application to support internationalization. Here you will learn more about the handling of numbers and date values in addition to the pure translation of strings.

The last three chapters deal with different environments touched by a React application. **Chapter 18** begins with an introduction to server-side rendering to improve the performance of an application. In this chapter, you'll also get to know the popular React framework, Next.js. **Chapter 19** is dedicated to various performance aspects of React applications. You'll then learn how to use React to implement a progressive web app that can be installed on user systems and run on as many systems as possible (**Chapter 20**). The concept of React on different systems is taken a step further in the final chapter, **Chapter 21**, with a focus on React Native. You'll learn how to use this library to implement native apps via React.

Download the Code Samples

All code samples in this book are available for download from the publisher's website at <https://www.rheinwerk-computing.com/5705>. Should you have any problems with the implementation, or if I have overlooked an error despite careful checking, please feel free to contact me at react@sebastian-springer.com.

Acknowledgments

I would like to thank all the people who helped me write this book. First and foremost, there's Philip, who once again took care of the review of one of my books and contributed many comments and tips.

A big thank you also goes to Friederike Daenecke for the linguistic polish.

I would also like to thank the entire team at Rheinwerk Publishing and especially Patricia Schiewald for her support.

Finally, I am very grateful to my wife Alexandra for all her patience and support.

Sebastian Springer

Aßling, Germany

Chapter 1

Getting Started with React

Surely you know Facebook, which is currently the largest social network in the world. Meta, the company behind this network, has developed a JavaScript library for the implementation of the platform to deal with the requirements of such a product. The biggest problem that arises when implementing a social network is the large amount of data, its display in the browser, and its updating. Because the frameworks and libraries available on the market in the past didn't meet Facebook's requirements, or did so only insufficiently, the company decided to write its own library—and this was the birth of React.

1.1 What Is React?

React is a library for implementing web frontends. However, React only covers the view layer—that is, the representation of the user interface. Regarding the structuring of the business logic and other architectural relationships, the library does not make any specifications, which results in a higher degree of freedom in the design of frontends compared to other frameworks and libraries. This freedom has the disadvantage that it is difficult, especially for beginners, to learn how to use React and to structure larger applications well. This comparatively high entry hurdle, combined with a steep learning curve, is also one of the biggest points of criticism of React. However, the initial extra effort in learning quickly pays off in development. The official website of the project, with a lot of further information and a tutorial, can be found at <https://react.dev>.

Getting started in the world of React is made much easier with tools like Create React App. Like many other JavaScript libraries and frameworks, React is an open-source project. Its developers have chosen to use the open-source MIT license, which is fairly open as it allows private and commercial use, requires the license and copyright to be stated, and excludes any liability. The source code of the project is maintained on GitHub and is available in the form of NPM packages. NPM is currently the largest package manager worldwide. It is used for almost all JavaScript projects and provides a public repository and a command line tool for package management.

React was developed with the ulterior motive to have the library able to be used on multiple platforms. This approach also explains why the library is divided into several packages. Thus, platform-specific components such as the renderer, which is responsible for rendering the application, can be replaced with little effort, while the rest of the

structure remains intact. In this chapter, you will get to know two essential parts of React in more detail, the renderer and the reconciler, which take care of finding the differences between two application states in order to render the application in a manner that is as performant as possible.

1.1.1 Single-Page Applications

Typically, you use React to implement single-page applications. The key concept of this type of application is that the browser does not have to be completely reloaded when a new state of the application is to be displayed. Hence the name *single page* as the application basically consists of only one HTML page. This page is customized by using JavaScript to reflect the current state of the application. However, this type of web application has the disadvantage that the initial loading process takes much longer than with a traditional website. The reason is that you need to load all the resources required to render the application. However, there are established solutions to this problem, namely lazy loading and server-side rendering, which you will get to know later in this book.

For the users of a web application, the single-page approach is much more convenient than a multipage approach, where the individual views of the application have to be reloaded each time the page is changed, as the transitions between the views can be designed much more smoothly. For a React application, a multipage architecture is only an option in rare, exceptional cases, because the entire source code of React must be loaded for each loading process. This problem can be minimized by using the browser cache so that only the first loading process takes more time and all subsequent ones run much faster. However, after loading the source code, React must be executed and must build the visible structure of the application.

In addition, a reload in the browser results in the current state of the application being discarded in the frontend and then having to be rebuilt. For a React application, this would mean that the state of all components would need to be stored on the server side or in the web storage of the browser. In a single-page application, the state of the application is preserved during a browser session, and you can access the contents of the memory and place objects there.

React exploits the characteristics of a single-page application to gain performance, which makes you feel that web applications are pretty much like native applications. However, by the time React reached the level of optimization it has today, the library had received numerous improvements and enhancements.

1.1.2 The Story of React

Compared to its competition, React is in the middle if you look at the appearance of the initial releases. The first version of Angular was released in 2009; Vue.js was released in

2014. React has been used at Facebook since 2011 and has had a colorful history since then. Originally, it was used for the central element of the social platform, the news-feed. Then in 2012, React came to Instagram, which was acquired by Meta. The decision to use React also contributed significantly to the abstraction of the library.

The Emergence of React

The history of React begins in 2011 under the name FaxJS. Jordan Walke developed a prototype of React using this library. One of the core elements of FaxJS was seamless rendering, whether server-side or client-side; even that early version was already environment-independent. In addition, responsiveness played a prominent role. When changes are made to an application's data, the frontend should automatically adapt. Fast content rendering and low load times were another feature of FaxJS. Similar to React, the library took a component-oriented and declarative approach to building applications graphically. FaxJS, in turn, was inspired by XHP, an HTML component framework.

In 2015, another platform besides the browser with native apps was supported by React: React Native.

The Jump from Version 0.x to Version 15

React follows the *semantic versioning approach*, where the version of a piece of software consists of three version numbers: major version, minor version, and patch level. For a long time, React was developed in version 0.x as part of minor updates. This practice is quite common in open-source projects and is meant to indicate that the project is in its early stages. During this time, breaking changes are also to be expected for minor releases, which, according to semantic versioning, may only be the case when the major version is increased. Overall, a 0.x version stands for limited stability. This approach has proven successful in other projects, such as Node.js. Again, the project was in this development phase for several years. The jump to a major version, as in the case of React to version 15, is intended to signal to users that the project has reached full production maturity and stability.

React has always been used live on the Facebook platform. There, the new versions are often used before the actual release so that they have already been extensively tested in practice at this point. This and the many years of use in production systems up to this point were to be further underscored by the leap to version 15.

The move to version 15 in April 2016 resulted in some significant changes to React itself. For example, support for Internet Explorer 8 has been removed. The development process became more transparent with the introduction of Requests for Comments (RFCs). Since the introduction of this development process, a change or change request for React has been published as an RFC and made available for discussion. These RFCs can be found on GitHub at <https://github.com/reactjs/rfcs>. In addition, and to improve transparency, the meeting notes of the core React team are published.

The most important technical innovation in React 15 was the significantly improved handling of the browser's DOM. In this context, the `createElement` method was used instead of the previously used `innerHTML` method. Also, the need to wrap text in `span` elements was removed, and Scalable Vector Graphics (SVG) support was completed.

React 16: The Next Big Leap

For a long time, the React community was eagerly awaiting the release of this version. The heart of this version was the new Fiber reconciler. A *reconciler* is the algorithm that calculates the differences between the current and the next version of the graphical interface. Fiber was supposed bring a significant performance boost and, more importantly, pave the way for future developments and new features. This became necessary because the stack reconciler reached its limits, especially with animations. Websites like <https://isfiberreadyyet.com> illustrate the mood at the time. `shows` how the website looked on March 26, 2017.

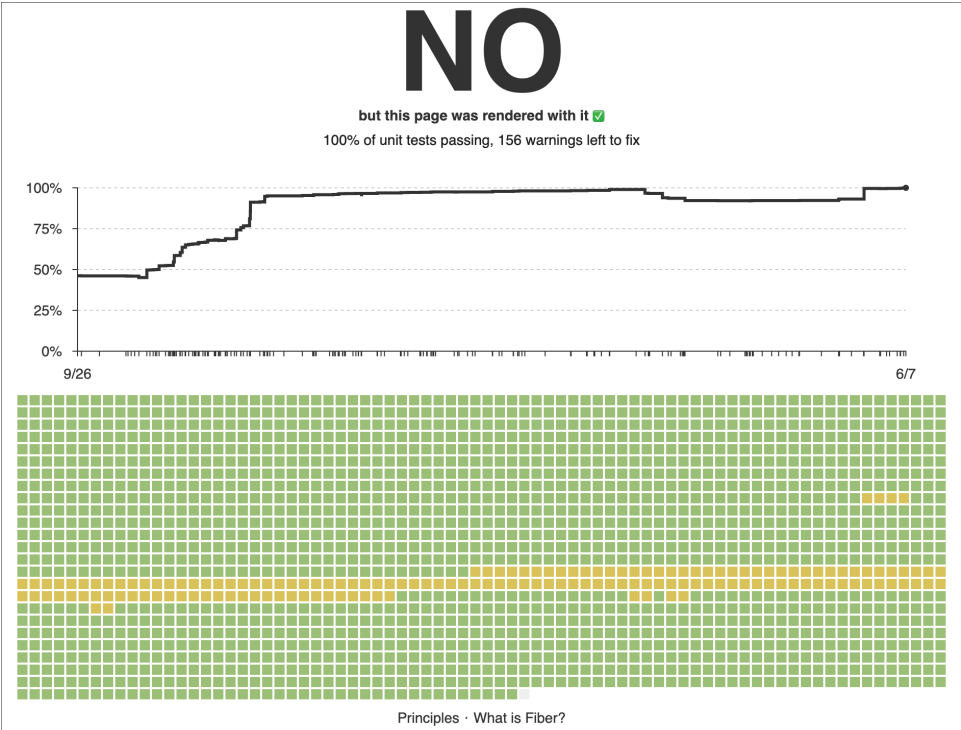


Figure 1.1 <https://isfiberreadyyet.com/> on March 26, 2017

The website answers whether the new release is already available with a prominently visible `No`. You can also see the status of the library's unit tests.

On September 26, 2017, the time had finally come: React 16 was released to the public as the new stable React version.

With the new reconciliation algorithm, whose functionality we'll describe in more detail in this chapter, and subsequent minor releases, numerous improvements and enhancements have been added to the core of React:

- **React 16.2**

The lifecycle of components up to this version was not oriented to the fact that the creation of the component tree could be canceled. New lifecycle methods were supposed to make this possible. A possible reason for such an abort is, for example, a higher-priority change, as is used in animations. A second feature worth mentioning in this release is the finalization of the Context API, which is used to exchange information between components across the tree structure. This interface was rebuilt with some adjustments to make it more convenient to use.

- **React 16.4**

One of the most important features of React is that the library is platform-independent, so it can be used on mobile devices as well as desktop systems. To accommodate this, support for pointer events was added to the library. Thus, touch surfaces and pens are now also natively supported as input devices.

- **React 16.6**

An important topic in React development is the performance of the library. With features like the `lazy` function, the first part of the suspense feature was implemented. This feature enables asynchronous content reloading. In the case of `lazy`, components can be loaded asynchronously and a placeholder can be displayed in the meantime. The static `contextType` property of a component makes accessing the context API even more convenient.

- **React 16.8**

In this release, the Hooks API was introduced. However, behind this feature, which at first seems inconspicuous, lies a significant upgrade of React's functional components. Using Hooks, it's possible to implement both the lifecycle of the component and its own local state in a function component, which before this feature was reserved only for class components. You can learn more about Hooks in [Chapter 6](#). This feature takes the decoupling of logic and state from the representation one step further. Despite the enhanced capabilities, the introduction of the Hooks API did not bring any breaking changes, so this extension also joined a series of minor updates that follow the roadmap of becoming a more performant and usable library.

- **Version 16.x**

Another feature aimed at improving performance is the *concurrent mode*, also known as *async mode*. This feature allows you to calculate the component hierarchy of the application in a separate process without blocking the main browser process. The `lazy` function implemented the first part of the suspense feature, which allowed reloading components. The *Suspense for Data Fetching* feature ensures that other server requests, such as asynchronous `fetch` calls to read data, can be wildcarded.

During the development of the new features, especially the concurrent mode, the

development team had to realize that the original roadmap was planned too optimistically. The team wanted to thoroughly test the new features first before including them in the stable release. For this reason, the concurrent mode was in an experimental stage for a long time, where it had to be activated separately.

React 17: The Featureless Update

The semantic versioning approach provides that a major update may include breaking changes. The release of version 17 of React was particularly surprising to the community, as it was announced with the headline “No New Features.” The focus of this release was to facilitate upgrades to new version numbers in existing applications. With this release, the development team introduced *gradual upgrades*, a feature that allows you to upgrade your application to a new version bit by bit.

Another change under the hood was that for event handling, React no longer registers its event listeners at the document level, but at the root node of the app.

With *server components*, React moves to bring the client and server closer together. This feature is intended to combine the interactive nature of React components as they are used in the browser with the performance of the server side. Like most other major new functionalities, it was initially added to the library as an experimental feature.

React 18: Concurrent React

The most important feature of the 18th version of React is the *concurrent renderer*. This feature allows React to prepare multiple versions of the graphical interface. This renderer changes the principles of the previous process and allows the library to interrupt the rendering process, continue it later, or start a new process. For the users of an application, this results in a more responsive UI, as the application can react more immediately to user interactions.

In addition to the concurrent renderer, React 18 also brings Suspense for Data Fetching for external frameworks, such as Next.js.

If you look at the development of React, it quickly becomes clear that the focus is strongly on performance, but also on good usability of the interfaces. The development is driven not only by Facebook itself, where React is strategically used to implement web frontends, but also by a very large and active community that has formed around the library.

1.2 Why React?

In client-side web development, there has long been an emotional discussion about which framework is the best. In addition to the three major solutions, Angular, Vue, and React, there are also numerous smaller solutions to choose from, such as Svelte. As

in other areas of software development, there is no clear winner, there is no right or wrong, but only alternatives. And so React is “only” an alternative to its competitors.

However, React has a very specific goal with its features, architecture, and philosophy. The focus of React is on user interface design, so when developing, the library gives you the freedom to design all other aspects of frontend development yourself. Like the other frameworks, React is built on a component architecture, but React components are very lightweight: in the simplest case, components are functions with a specific return type.

Another feature of React is that there is no predefined architecture like there is with Angular. Concepts like dependency injection or services are foreign to React. You have the freedom to design your architecture as you need it for your application, and React supports you in this but does not impose strict requirements. However, this fact is not always an advantage as it makes it difficult for beginners to find their way in React. For this reason, both the React team and the community strive to make it as easy as possible for new developers to learn React and use the library in production systems.

React follows the semantic versioning approach to versioning. Unlike other open-source projects, however, React does not provide for a regular release cycle in which, for example, new major releases are published every six months.

Semantic Versioning

In the *semantic versioning* approach, the version number of a project is divided into three numbers separated by a period. This results in a version number in the format X.Y.Z. The versioning usually starts with 0 and has no upper limit.

The first number of the version number is called the *major version*. This number is increased in the case of *breaking changes*, changes that ensure that the application can continue to run only with adjustments to the source code. One example in React is version 16, which introduced the Fiber reconciler, among other things.

The second number of the version number, which is also referred to as the *minor version*, is incremented for new features. For example, in version 16.8, Hooks were introduced as an extension to the function components. The important aspect about a minor release is that the application must still be able to run without changes in this case.

The last number, the *patch level*, is increased for bug fixes. Such releases occur comparatively often and usually do not cause any problems.

React developers are careful to keep breaking changes, meaning major updates, to a minimum. This is also reflected in the frequency of recent major releases: React 15 was released in April 2016, version 16 was released in September 2017, React 17 in October 2020, and finally React 18 in March 2022.

If changes to the interface are pending, such as in the lifecycle of the class components, then they are not introduced directly in the next version and the previous version is no longer supported. Instead, in such a case, the previous methods get renamed and are still available for a transition period. For such customizations, the developers of React provide *Codemods*, which is a tool for the automated customization of the source code. We'll present this tool in more detail in [Chapter 2](#). In addition, the React team uses *deprecations*. This means that the team marks parts of the library that will be removed in the future, so that everyone who develops applications is warned and can schedule appropriate refactorings.

1.3 The Most Important Terms and Concepts of the React World

Development of a React application is sometimes very different from that of traditional web applications. For example, React follows a component-oriented approach to building an application. This means that an application is composed of numerous small building blocks that are loosely connected to each other. In addition, React has some optimizations that allow changes in the appearance of the application to be displayed in a very performant way. The goal of React is to be able to handle very large amounts of data in the frontend.

1.3.1 Components and Elements

As mentioned, the central elements of React are *components*. They are the building blocks of an application and provide the structure of the interface. A component should be as small as possible in its scope and independent of its environment. This makes it possible to use it in multiple places in the application or even in multiple applications. The core of a component is a *function* that returns the structure of the component. This function is referred to as the `render` function.

You can define the structure of the display in two different ways:

- You can use the `React.createElement` method to create new elements and build a hierarchy with those elements.
- It's also possible to use JavaScript XML (JSX)—a syntax extension for JavaScript—and thus use HTML-like notation directly in the JavaScript source code of your component to describe the structure.

The more common variant, which is used throughout this book, is the second one using JSX.

A component is not the smallest unit available to you in React. At the lowest level, there are the *React elements*. These are translated into native elements, depending on the environment. For the browser environment, this means that the React `div` element is

translated into an HTML `div` element. In a component, you can use both components and elements to describe the structure of your application. To distinguish between elements and components, React has introduced the convention that elements always start with a lowercase letter and components with an uppercase letter.

Components can be parameterized for improved reusability. If you use JSX, you can pass attributes to a component that you write as a JSX tag. These are referred to as *props* in React and allow objects to be passed in the component tree.

In general, React components have a defined lifecycle that you can intervene in at various points to influence the behavior of a component. Components can also have their own state. This *state* is a data structure that contains information for representing the component. If the value of the state changes, React makes sure that the change is reflected in the browser and the component will be rendered anew.

When it comes to components, React again distinguishes between two categories: class and function components.

Class Components

The class components of React are in some ways relics from a bygone era. They represented the standard when it came to components prior to the introduction of the Hooks API in version 16.8 because only class components had their own state and lifecycle methods. However, the introduction of the Hooks API has caused class components to become less important. In modern React applications, class components are hardly ever found.

The reason for the decline of class components is that the second type of components, *function components*, are more in line with the character of React. They are more lightweight and flexible. In addition, the Hooks API solves some problems of the class components. However, we'll go into these aspects in more detail in [Chapter 6](#).

The name *class component* is based on the fact that a class component is a JavaScript class that derives from the `React.Component` base class. In previous versions of React, class components were created using the `React.createClass` method, although this method is now no longer available and you should only use the class-based variant. Alternatively, you can use the `create-react-class` add-on package, which is an interface that behaves very similarly to `React.createClass`.

The core of the class component is the `render` method, which makes sure that the component can be displayed. The method returns a React element or component as a return value that React uses for rendering. React manages the state of a class component in the form of a property and maps the lifecycle through various methods, such as `componentDidMount`.

When you start implementing a new React application, you should avoid using class components and instead draw on the more modern function components.

Function Components

As the name suggests, a *function component* consists of a single function. This function is nothing more than the `render` method of a class component and takes care of rendering the component. By default, such a function component originally did not have its own state and lifecycle methods. Function components were used in their original form as simple display components. However, with the introduction of the Hooks API, this changed and function components became full-fledged React components with state and lifecycle and have now almost completely replaced class components. With a few exceptions, this book focuses exclusively on function components for developing applications.

1.3.2 Data Flow

A React application lives by its dynamics. As a rule, users interact with your application and use it to produce and consume information. A core element of building a React application is modeling data streams. In the component tree of the application, information always flows from the parent elements toward their children.

If you imagine a typical React application that is used to manage information, typically one of the first requirements is to implement a list representation of the information. In the component tree, the list itself is represented by a component. The individual entries are in turn components that are in a parent-child relationship with the list. The list entries are the child elements of the list (see [Figure 1.2](#)). To display the information, you read the data into a central point, such as the list component, and distribute the information to the individual child elements. This solution has the decisive advantage that only a summarized request is required, and you receive an overview immediately.

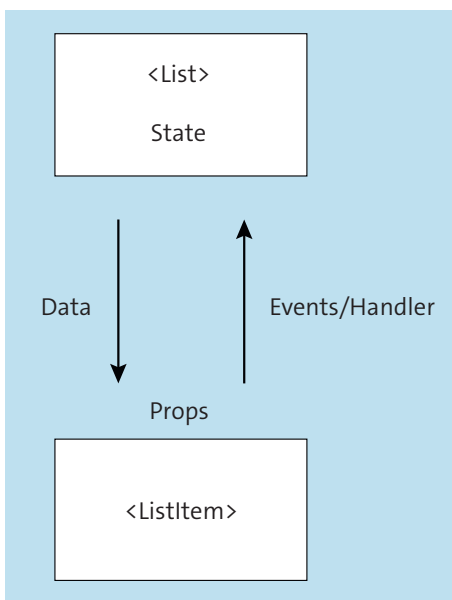


Figure 1.2 Data flow in React Application

The `list` component contains the state; that is, it is responsible for data management. It downloads the data for display from the server and takes care of sending changes to the server. For each data record, a child component is created, which receives the information to be represented as props. This method makes sure that a directed data flow can be implemented, through which React is able to make the rendering of the interface very performant. A side effect of this component partitioning is that you can use the components in multiple places in your application.

However, with this type of data flow, the child components have no way to pass information back to their parent components. But that becomes necessary when information has been modified and the parent component should be notified about it. One solution to this is to pass callback functions from the parent to the child component in addition to the rendering information. These functions, known as *event handlers*, are executed by the child component in certain cases—for example, when data is changed. The function works in the scope of the parent component, so it has access to its internal structures and can, for example, modify the state. This gives you the option to return the information to the parent component.

The performance benefits that are caused by the directed data flow come from the fact that React knows exactly where to change the application's display.

1.3.3 The Renderer

In the default configuration of a React application, the `react` and `react-dom` packages are installed. The `react` package contains the library itself. The second package, `react-dom`, contains the renderer. This component is used to translate React elements into concrete HTML elements that can then be rendered in the browser.

Another renderer for React is *React Native*, which ensures that the elements of a React app are translated into their native equivalents in an iOS or Android app. If you use different renderers, you should note that the different environments use their own elements. For example, you can use `div` elements with `react-dom`. In React Native, you use the `view` element as the container element instead.

Not only can a React app be rendered in the browser or on a mobile device, but there are many other renderers available as NPM packages. For example, Ink is a renderer for interactive command-line applications using React.

1.3.4 The Reconciler

Among other things, the main React package contains the reconciler. This algorithm is responsible for detecting changes in an application. An adjustment to a component can be made either by changing the props or the state. For a change to take effect in the browser, the DOM tree must be at least partially rebuilt in the case of the browser environment. The rendering of HTML structures is still one of the biggest weak points for the performance of web applications. React provides a solution to this problem in the

form of the *virtual DOM*, an image of the application's structure in the form of JavaScript objects.

If a modification of the DOM structure is required, React generates a new version of the virtual DOM. This structure serves as a blueprint for the new version of the application. React then attempts to customize the existing DOM using a series of optimized actions. For the optimization of the reconciler algorithm, two assumptions are made to reduce complexity:

- Two elements with different types produce different trees.
- The key prop can be used to specify which child elements remain stable between two render steps.

When comparing the original state and the target state, React proceeds from the root toward the child elements and performs the comparisons.

As already mentioned, the reconciler first checks whether the types of two elements match. If the types are different, the first optimization of the algorithm takes effect, which states, as mentioned earlier, that two different types lead to two different trees (see [Figure 1.3](#)).

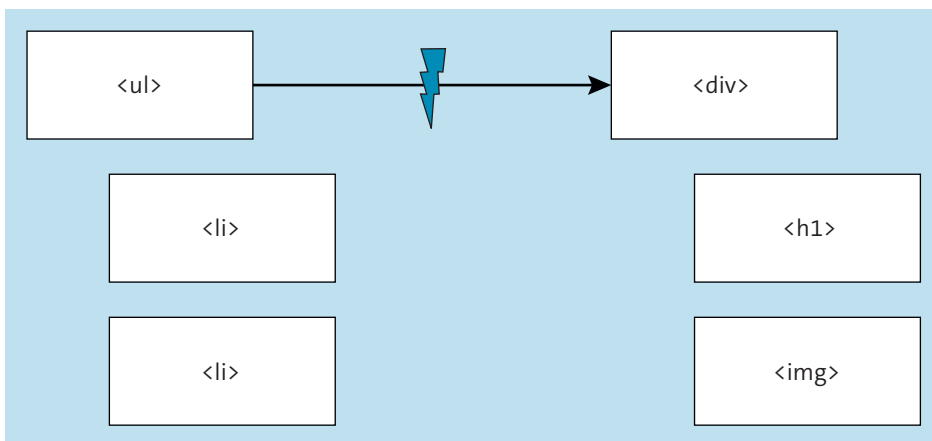


Figure 1.3 Different Tree Structures

React can abort the check in this case and build a completely new subtree. This means that all previously built structures are discarded and all components are unhooked. This terminates the component lifecycle and, if present, executes the appropriate unmount logic used to clean up the environment. The new components are then built and the appropriate lifecycle functions are executed.

If the types of two elements match, then the attributes of the elements are checked and only the values are adjusted accordingly. For example, if the element's class name or style attribute changes, the underlying DOM element is adjusted accordingly and the subtree is not discarded (see [Figure 1.4](#)).

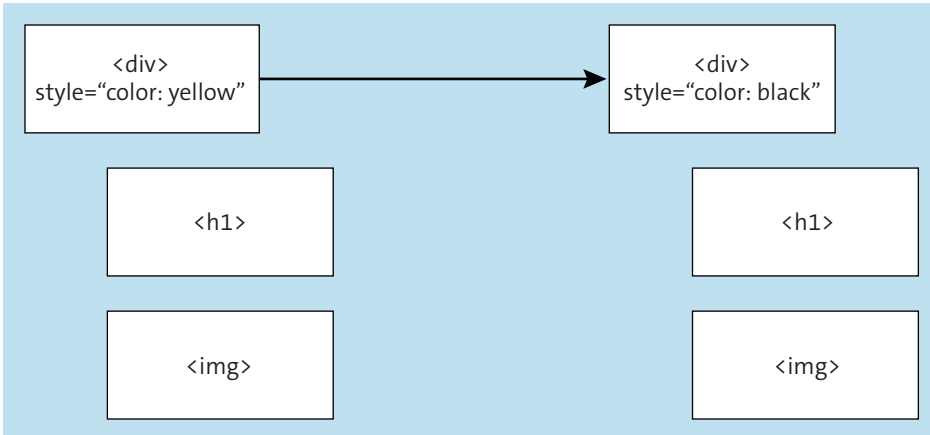


Figure 1.4 Modification of Attributes in Tree Structures

With components, the situation is similar to elements with changed attributes: The component instance remains the same, so the state is preserved and the corresponding lifecycle functions are executed. Then the child elements and components are processed.

Multiple Child Elements and the “key” Prop

In the case of a list of several elements, it may happen that only the order is changed, but not the elements themselves. This becomes relevant, among other things, when a new element is inserted at a certain position in the list. As a result, the index of all subsequent elements changes, which would mean for React that they have to be rebuilt. This can be prevented by means of the `key` prop. The `key` prop must contain a unique and stable value in a list. If the `render` method of the parent component is executed again, the value of the `key` prop is used to check whether the elements have changed.

Typically, the unique IDs of data records are used as the value for the `key` prop. The value does not have to be numeric. However, it is mandatory that the value is unique within the list, not across the entire application, and does not change between the different `render` calls; otherwise the assignment cannot be done correctly.

As an aid during development, React issues a warning at each iteration about a list of elements without a `key` prop. This warning states that each child element of a list should have a unique `key` prop. Especially for large lists, using the `key` prop can significantly improve the rendering performance.

These concepts have given you a first glimpse into the world of React. The following chapters of this book build on these terms and provide additional detailed explanations. In the next section, you’ll learn about some other libraries that are often used in conjunction with React.

1.4 A Look into the React Universe

React is a specialized library for creating graphical interfaces in web frontends. Unlike full-featured frameworks (such as Angular), such libraries cover only a specific aspect of an application. The more extensive an application becomes, the more additional libraries must be included to speed up development and keep the source code manageable. In the following sections, we'll briefly introduce some of the most important libraries in the React universe. In the course of this book, you'll get to know these tools in greater detail.

1.4.1 State Management

One of the most popular architectural forms of large React applications is the *flux* architecture, which provides for the decoupling of the presentation from the state and the business logic. A concrete implementation of the flux architecture is the *redux* library. At its core, redux provides a central store for storing the application's information. The data of the store can be read, but not directly written. Here you have to take a detour via so-called actions. These are simple JavaScript objects that describe the changes. They are received by reducer functions, which in turn can modify the store. To learn more about centralized state management and redux, see [Chapter 14](#).

1.4.2 The Router

If the application has multiple views, switching between them can be quite time-consuming and result in cluttered code in the application. A solution that is also available in most other frontend frameworks is *routing*. This feature refers to an extension that can be used to insert component trees depending on the selected URL. In the case of the React router, the history API of the browser can be used.

In addition to just navigating between component trees, the router supports other features, such as variables in the URL that you can access in the components, or nested routes.

1.4.3 Material UI

You can find numerous design recommendations on the web. One of the most widely used is Google's Material Design. So that you don't have to implement the individual elements yourself, the Material UI package is a collection of components that implements the recommendations of Material Design. The component collection includes not only standard components such as buttons or input fields, but also more extensive components such as dialogs, data tables, or menus.

1.4.4 Jest

Meta has developed *Jest*, a testing framework that, while not directly tied to React, is ideally suited for use with the library. By default, Jest does not require any additional configuration and runs tests in a simulated environment rather than in the browser. This fact also ensures that the tests are executed much faster compared to other frameworks (such as Jasmine in combination with Karma).

Another notable feature of Jest is snapshot testing, which allows you to create a static representation of a component and use it as a basis for comparison.

1.5 Thinking in React

In the React documentation, you'll find a section called "Thinking in React" (<https://react.dev/learn/thinking-in-react>). It describes how you should proceed when creating an application. Here, React's component-oriented approach plays a prominent role, resulting in an application consisting of a tree of components. You can take advantage of this fact during the building phase and follow an overall five-step process.

The development process always begins with a concrete idea: a simple mock will suffice here. As a concrete example, examine the form in [Figure 1.5](#).

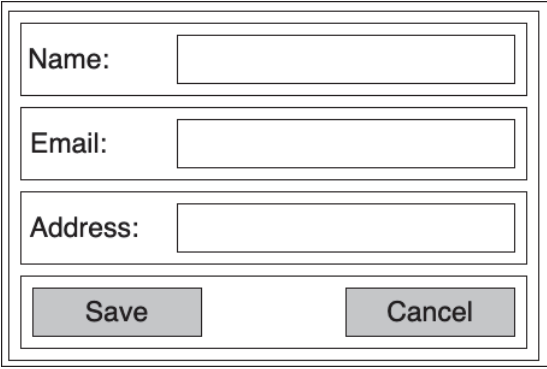
A diagram of a web form titled "Draft Form". The form is enclosed in a double-line border. It contains three input fields, each with a label to its left: "Name:", "Email:", and "Address:". Below these fields are two buttons: "Save" and "Cancel". The buttons are rectangular with a slight shadow and are positioned side-by-side.

Figure 1.5 Draft Form

1.5.1 Decomposing the UI into a Component Hierarchy

A *component* is a building block of an application. In the simplest case, you can represent the boundaries of the components in your mock by rectangles. In doing so, you should proceed from the larger components to the smaller ones.

A component should only perform one task. For example, you can divide the form into its individual sections and these in turn into the description and the form field.

1.5.2 Implementing a Static Version in React

Then you can statically implement the components identified in the previous step. This creates a collection of components, which you can use in the further development of your application.

The individual components can be nested and the information passed on via props within the component hierarchy.

1.5.3 Defining the Minimum UI State

The state of the component hierarchy contains both the data to be displayed and the state of individual visible parts of the component. When modeling the state, make sure you don't produce any duplicates. These must be synchronized during the runtime of the application and carry the risk of errors and inconsistencies.

1.5.4 Defining the Location of the State

Now that you know what data you need, you still need to determine the appropriate location for it. As a rule, you should place the state where it is directly needed. An exception to this is when you place the state at a parent location to pass the information to multiple child components.

1.5.5 Modeling the Inverse Data Flow

The information from the state is passed to the child components via props. If it should be possible for the child components to modify parts of the state, you must implement this by means of functions. In this case, you pass functions from the parent to the child components in which you modify the state of the parent components.

You can even build your entire application according to this scheme. Over time, you can build your own library of components that you can reuse, which speeds up your development work as a whole.

1.6 Code Examples

In this book, you will become more familiar with each area of React and the ecosystem that has formed around the library. Reading the theory is one thing, but applying what you've read in real life is completely different. Only when you work with React and its numerous extensions yourself will you really get to know the library. The examples in each chapter are self-contained, so they do not build on each other and function independently.

You can implement almost any application in React, from classic CRUD applications for managing data records to complex interactive applications that you can use to map business processes to browser games. The individual chapters, while independent, share a topic that will run throughout the book. The examples represent the parts of a library—that is, an administrative interface for books, which enables you to view, create, modify, and delete data records.

1.7 Summary

This chapter presented an introduction to the world of React. The main topics you learned in this chapter are as follows:

- React is usually used in the context of single-page applications.
- The development of React spans from the initial introduction to Facebook to the current development and integration of the concurrent renderer and server components.
- React follows the semantic versioning approach, where breaking changes only occur in major versions. However, the development team tries to avoid such breaking changes if possible.
- Components are the building blocks of a React application. They come in two flavors in React: as the old class components and the more modern function components in combination with the Hooks API.
- You can model the data flow using props by passing objects and functions to child components. The functions allow child components to notify parent components.
- In React, the renderer takes care of the rendering in the respective target platform.
- The reconciler is the algorithm used to calculate the differences between the current component tree and the future component tree.
- You also learned about some libraries that are used together with React in an application.
- Finally, you learned how to proceed when building an application.

This chapter was mostly concerned with theoretical concepts behind React. In the next chapter, you'll learn how to start developing a React application.

Chapter 2

The First Steps in the Development Process

In this chapter, you'll learn about the development process: from initializing the application, configuring the development environment, and debugging the application to building the application.

The first steps in developing a single-page application usually involves a lot of effort: you need to download and install dependencies, create structures—that is, files and directories—and either launch the application directly from the file system or deliver it via a web server. In this chapter, we'll look at the lifecycle of a React application. You'll also learn about the different ways to start developing such an application.

However, you'll usually use an established and very widely used tool called *Create React App* to start your development work. This command line tool takes care of the most important steps and creates a new React application for you, which enables you to start developing immediately.

2.1 Quick Start

Do you want to start developing your application immediately and deal with the background and the various alternatives later? You'll learn how you can do that in the following sections. For explanations and details, just continue reading the chapter to the end.

2.1.1 Initialization

To initialize a new React application, first switch to the command line of your system and enter the following command:

```
npm init react-app library
```

Listing 2.1 Initializing an Application Using NPM

A prerequisite for the successful execution of this command is a local installation of Node.js including NPM. The `npm init` command creates a new directory named *library* where your application is located. All dependencies and structures are prepared to the point that you can start your application with the commands from [Listing 2.2](#) and start developing.


```
cd library  
npm start
```

Listing 2.2 Starting the Application

More information about Node.js and NPM, the Node package manager, will be provided in [Section 2.4.2](#). At this point, all you need to know is that NPM is part of the Node.js platform, which you can get from <http://nodejs.org>.

2.1.2 TypeScript Support

I recommend that you always initialize your React application, no matter how small, with TypeScript. For this purpose, Create React App provides application templates. For example, the command

```
npm init react-app library --template typescript
```

Listing 2.3 Initialization with TypeScript

integrates TypeScript and all necessary tools into the application during the initialization. Then you can switch to the directory and start developing.

2.2 Playgrounds for React

To quickly try something with React or get an initial feel for the library, you don't necessarily need to install anything on your system. Numerous platforms exist that provide you with a basic version of React in the browser. For example, you can create small applications and test them directly in the same window. This approach is only suitable for a very small scale and for smaller experiments.

Warning!

As soon as you start working on a larger application, you should write the source code locally on your system and stop using such a playground.

Ahead, we'll introduce one of these platforms: CodePen.

2.2.1 CodePen: A Playground for Web Development

You can use the CodePen platform both after a free registration and anonymously without registration. The address of CodePen is <https://codepen.io>. As with other similar platforms, you have three editors, one each for HTML, CSS, and JavaScript. Also, the

platform will show you the result of executing your code in another section of the window. You can vary the size of the individual sections using drag and drop, and you can also customize the layout of the entire platform.

Users that are logged in can save their experiments and have an overview of the saved projects through a dashboard. The registration can be done either directly via a CodePen account or via a Twitter, GitHub, or Facebook login.

[Figure 2.1](#) shows the default view of CodePen. In the next section, you'll learn the steps to get to a React playground where you can run your experiments.

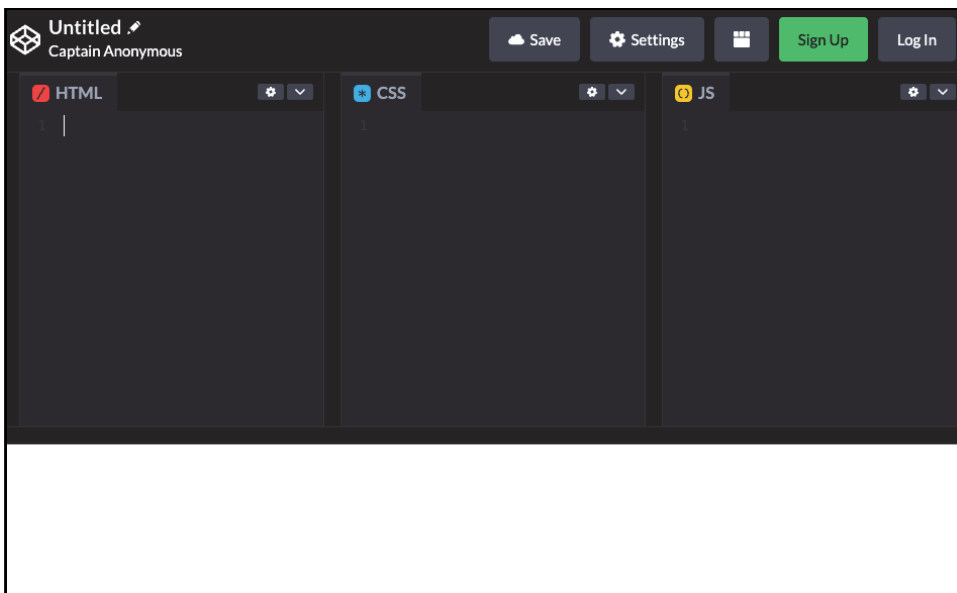


Figure 2.1 CodePen View

2.2.2 A React Project on CodePen

In the default configuration, CodePen doesn't include any libraries and just provides you with a combination of HTML, CSS, and JavaScript. You can activate React in this environment by first clicking the **Settings** icon in the JavaScript editor. There, under the **JS** item, select **Babel** as the **JavaScript preprocessor** and then add the two libraries, **react** and **react-dom**, under **Add External Scripts/Pens**. This configuration allows you to start developing your React application.

Babel

Babel is a JavaScript compiler that accepts JavaScript source code and in turn also produces JavaScript source code. The purpose of Babel is to simulate features that aren't yet implemented in certain browsers. Babel itself only provides the compiler infrastructure

and can be extended by plugins. Each of these plugins is responsible for a specific aspect, such as supporting arrow functions. Several plugins can be combined into *pre-sets*. This is a convenient feature that should save you from having to include many plugins. For example, the *React preset* groups some JSX plugins for you.

A React application requires an HTML element in which to insert the application. For this purpose, you need to insert a `div` element in the HTML editor with an `id` attribute and the value, `root` (see [Listing 2.4](#)). The value of this attribute is freely selectable and may differ. In this case, you must adjust the reference when you bootstrap the application.

```
<div id="root"></div>
```

Listing 2.4 Container for the React Application

In [Listing 2.5](#), you can see the JavaScript source code of the sample application. It consists of defining a component and rendering the React application.

```
const Greet = ({ name }) => <h1>Hello {name}!</h1>;

const rootElement = document.getElementById('root');
const root = ReactDOM.createRoot(rootElement);
root.render(<Greet name="Reader" />);
```

Listing 2.5 Source Code of the React Application

In the JavaScript editor, you first create a simple component called `Greet`. Make sure that the name of the component starts with an uppercase letter. The component is an arrow function that returns a JSX element. The function receives a props object as an argument, which you can use to access passed values. You can access the `name` value, which you extract via a destructuring statement, in the JSX structure with curly brackets. The details about the structure of a component can be found in the following chapters.

After defining the component, you get down to the actual setup of the application. First, you need a reference to the container element where you are inserting the application—that is, the `div` element you defined earlier. Then you use the `createRoot` method to create a root object from the `ReactDOM` object and the reference to the `div` element. In the final step, you call the `render` method of the root object and pass it another JSX structure, which contains the `Greet` component as an HTML tag with the `name` attribute. This way you can make sure that React renders the component and produces output like that shown in [Figure 2.2](#).

The CodePen platform has a pretty simple structure. As a consequence, in the default setup, you have no way to divide your application into multiple JavaScript files. Other platforms, such as CodeSandbox, for example, offer this feature, but local development

of applications is still much more convenient and flexible. In the following section, you'll learn how to start developing your React application on your own system.

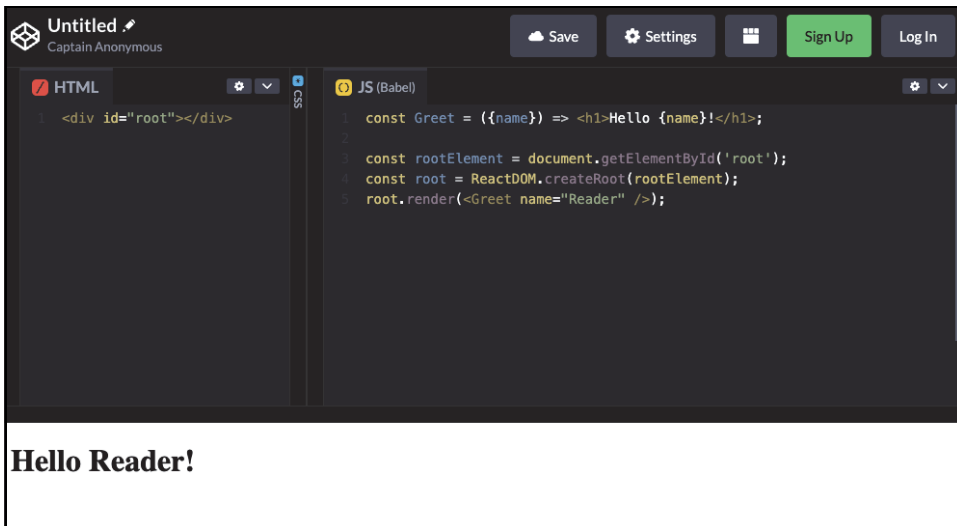


Figure 2.2 View of the React application in CodePen

2.3 Local Development

The advantage of a platform like CodePen that is available anytime and anyplace and in which you can share your code with other developers worldwide is somewhat marred by the disadvantage that debugging in such an environment is only possible in a pretty awkward manner. You also need an existing internet connection and have no way to keep your source code in a local repository. In most cases, you will develop your application locally on your computer. Once again, there are numerous options. The easiest way is to integrate the library directly into a web page.

Compared to libraries like Vue.js, React is often criticized for being comparatively difficult to get started with. Whereas Vue.js only requires you to include a JavaScript file, React seems to require you to go through an extensive installation process first. However, this is only partially true.

To use React in a small local application, you should follow a similar procedure to configuring a playground such as CodePen. First, you want to create an HTML file named *index.html*:

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8">
```

```
<title>Hello React!</title>

<script
  src="https://unpkg.com/react@18/umd/react.development.js"
  crossorigin
></script>
<script
  src="https://unpkg.com/react-dom@18/umd/react-dom.development.js"
  crossorigin
></script>
<script src="https://unpkg.com/babel-standalone@6/babel.min.js"></script>
<script src="index.jsx" type="text/babel"></script>

</head>

<body>

  <div id="root"></div>

</body>

</html>
```

Listing 2.6 Initial index.html File with Direct React Integration

Listing 2.6 shows the structure of this initial file. First, you need to make sure that the `react` and `react-dom` libraries are loaded. You can do this via the *unpkg.com* content delivery network (CDN). To use the more comfortable JSX syntax to build your components, you also need Babel, which you can also get from *unpkg.com*. With this setup, you can finally include the `index.jsx` file that contains your custom application. At this point, it's important that you specify the `type` attribute with the value `text/babel`. This way Babel can ensure that the file is translated so that the browser can interpret the JSX code. In the body of your HTML file, you define the container in which your application will be mounted. The `.jsx` file extension signals that this file is not an ordinary JavaScript file, but that JSX is used and therefore a tool like Babel is needed for translation.

The contents of the `index.jsx` file correspond to the source code that you executed as JavaScript in CodePen in the first example. Listing 2.7 summarizes it again for you:

```
const Greet = ({ name }) => <h1>Hello {name}!</h1>;

const rootElement = document.getElementById('root');
const root = ReactDOM.createRoot(rootElement);
root.render(<Greet name="Reader" />);
```

Listing 2.7 JavaScript Source Code of the Local React Application

When you load the *index.html* file locally in your browser, you will only see a blank page, and when you look at the JavaScript console of your browser, you'll see an error message telling you that the *index.jsx* file could not be loaded. This is because Babel tries to load this file by means of an asynchronous request, and for security reasons such requests cannot be executed via *file://*-, but only via the *http://* protocol.

The easiest way to fix this problem is to test your application with a local web server.

Using a Local Web Server

Some browser features require that an HTML document be read from a web server rather than from the local hard disk. To simplify this, numerous projects exist that provide you with a lightweight web server. One of the best-known projects is called *http-server* and is available as an NPM package.

To use the *http-server* web server, you have several options. You can install it locally, globally, or on demand:

■ Local installation

For the local installation, you need to switch to your system's console, navigate to the directory where your React application is located, and run the following commands:

```
npm init -y
npm install http-server
```

The first command creates a *package.json* file for your project, which contains its description and configuration, such as installed dependencies. The second command installs the *http-server* in the current directory in the *node_modules* subdirectory. After that, you can start the *http-server* using the *node_modules/.bin/http-server .* command. The server then delivers the contents of the current directory to all available network interfaces on your system, so you can access your application via *http://localhost:8080*.

■ Global installation

You can start the global installation on the command line via the following command:

```
npm install -g http-server
```

This makes sure that the package is installed into a directory that is in the system's search path so that you can run the *http-server* on the console using the *http-server .* command. Again, the program returns the contents of the current directory in all interfaces on TCP port 8080.

■ On-demand execution

The third and last variant consists of using the *npm* command, which has been part of the NPM since version 5.2. Using *npm http-server .*, you can first search for a local version of the package; if that can't be found, it will be downloaded and executed directly. Again, the contents of the current directory are made available. Once the package has been run, it will be discarded.

Which option should you use for which purpose? In general, you should avoid global installations of packages as this creates an implicit dependency and locks you into one version for your entire system. The local installation of a package is worthwhile whenever you need the functionality more than once, whereas the on-demand installation can be used when you need a functionality once or only a few times.

If you now run `http-server` as described, you'll reach your application via `http://localhost:8080`, where it should run without errors and you should get a view like the one shown in [Figure 2.3](#).

After these two digressions into the execution of React, in the following sections we'll focus on the variant most commonly used for working with a React application: the *Create React App* project.

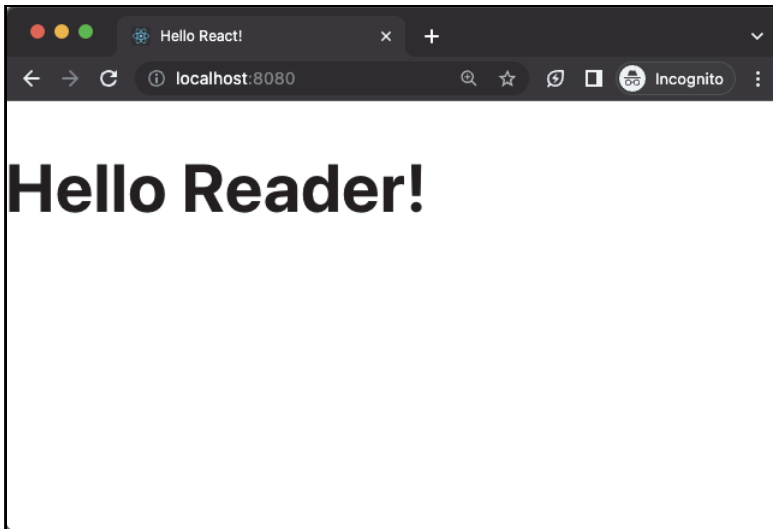


Figure 2.3 Local Execution of the React Application

2.4 Getting Started with Developing in React

Almost all major JavaScript frameworks have command line tools, which relieve you of routine tasks that arise in the course of development. The most time-consuming task in the development process is setting up the environment. This is mainly because numerous tools have to be put together for the development and build process.

Typically, tools such as package managers, bundlers like *Webpack*, *Babel* as a transpiler, and numerous others collaborate in the development of a web application. A command-line tool like *Create React App* handles the coordination of these tools and creates a basic structure for your application.

2.4.1 Requirements

To implement a React application, you need some tools on your computer.

Node.js

For example, you should have an up-to-date version of Node.js. You can get it either directly from <https://nodejs.org/> as an installer package or via the package manager of your operating system.

Node.js

Although React is a frontend library that can be used independently of Node.js, the server-side JavaScript platform nevertheless plays an important role in the development process. Many tools you use during the implementation are based on Node.js. Node.js is based on the V8 engine, which is also used in the Chrome browser. The platform has a set of core modules that allow you to access operating system resources (such as the network or disk) during development.

In addition to the JavaScript platform itself, the Node.js package also includes *NPM*, a package manager that can be used to manage the dependencies of your application.

In this chapter, you'll also get to know *Yarn*, a largely API-compatible alternative to NPM.

On the command line, you can use the following commands to check that the two tools are installed correctly:

```
$ node -v
V19.2.0
$ npm -v
8.19.3
```

Editor

In addition to Node.js, you should install an editor on your system that you can use to write the source code of your application. I recommend using either the free Visual Studio Code editor from Microsoft or the paid development environment WebStorm from JetBrains, but you can use most any editor. However, you should make sure that it has convenient features such as syntax highlighting for JavaScript, TypeScript, and, in the best case, React and JSX, and that it offers you *code completion*—that is, the correct completion of variable and method names when you have started typing them.

Browsers

React can be run in various environments. However, React applications are usually run in the browser. So for development purposes, you should have at least a recent version of one of the four main browsers: Chrome, Firefox, Edge, or Safari. The examples in this

book, insofar as they concern specific browser features, refer to Chrome because it's so widely used. Functionalities such as the developer tools can also be found in the other browsers in a similar form and to a similar extent.

React supports all modern browsers. Since version 18 of React, Internet Explorer is no longer supported. Microsoft discontinued support for this browser on June 15, 2022, anyway. If you still want to support older browser versions or even Internet Explorer, you need to install various polyfills. These are combined in the `react-app-polyfill` package and can be installed via a package manager. After that you only need to integrate the version suitable for your browser. [Listing 2.8](#) demonstrates this using Internet Explorer 11 as an example.

```
import 'react-app-polyfill/ie11';
```

Listing 2.8 Loading Polyfills for Internet Explorer 11

Polyfill

A *polyfill* is a piece of JavaScript source code that is used to emulate a feature that isn't present in the browser. Polyfills are usually much less powerful than the actual browser features. However, the main issue here is not to exclude any users, and less about performance.

2.4.2 Installing Create React App

The variants presented so far for starting development with React have the disadvantage that they allow a quick start in development, but the development comfort falls by the wayside. This problem is solved by the Create React App project. This is a command-line tool developed by Facebook. The project's website can be found at <https://create-react-app.dev/>. Unlike other command line interface (CLI) tools (such as Angular CLI) that guide you through the entire app lifecycle, Create React App only assists you in the app creation process.

Create React App takes care of almost all the work involved in initializing an application. The tool creates the basic structure of the application, downloads all the required dependencies, and prepares everything to the point where you can start developing directly. The generated application uses packages established in the community for standard tasks, such as Webpack as a bundler and Babel as a transpiler. However, the configuration of the tools used is handled by Create React App by default and hidden from you, so you don't come into contact with it. In most cases, the default configuration is also sufficient. If that isn't the case for your application, you have the option of exporting the configuration and adapting it accordingly. To learn exactly how this works, [Section 2.4.4](#).

Another convenient feature Webpack offers is the *Webpack dev server*, a local web server through which you can test your application directly. During development, each time a change is made to the source code, the affected files are reprocessed and inserted into the running application. The infrastructure ensures that the browser is automatically reloaded so that the changes take effect immediately. Under no circumstances should you use the dev server for production operations as reloading the application results in resetting the state of the application in the browser.

The primary goal of Create React App is to get you started quickly with development while keeping complexity to a minimum. This is noticeable on the one hand by the default configuration mentioned earlier and on the other hand by the handling of dependencies. If you look at the list of directly installed dependencies in the *package.json* file of a fresh React application created using Create React App, you'll find a total of three installed libraries: `react`, `react-dom`, and `react-scripts`. All other required dependencies are dependencies of these three libraries, which also greatly simplifies the update process of an application.

But enough of the introductory words; now let's start the development of an application! You have several options for installing Create React App, depending on which package manager and strategy you choose.

Initialization Using “npm init” and “npx”

While several package managers exist for JavaScript, NPM is by far the most widely used. The simplest way to initialize your React application is to use the `npm init` command. Using

```
npm init react-app library
```

Listing 2.9 Initializing a React application using “npm init”

you also create a local directory named *library* where Create React App builds the structure for your application. If Create React App isn't installed on your system, NPM will download it temporarily, run it, and then discard it. Typically, you use the `npm init` command to create a new *package.json* file in an interactive process. In our case, you also pass the `react-app` initializer. NPM then uses the `npx` tool to download the package named `create-` and the name of the initializer and run it. Alternatively, you can also use `npx` directly. In this case, the relevant command is as follows:

```
npx create-react-app library
```

Listing 2.10 Creation of a New React Application Using “npx”

The effects of both commands are equivalent, and you can start working on your application directly afterward.

The Node Package Manager

The name of the NPM tool is somewhat misleading. NPM is a package manager for JavaScript that you can use both for server-side development with Node.js and for managing dependencies in the frontend. NPM is developed as an independent open-source project and is part of most Node.js installations. On the command line, NPM is available across the entire system via the `npm` command.

Besides the `npm` command, there are other important components of NPM in your project: The *package.json* file, which is usually located in the root directory of your application, contains the description of your application. The *node_modules* directory stores the installed packages. Usually, this directory does not get versioned along with the application. Instead, if you want to rebuild an existing application from the version control system, you need to run the `npm install` command to install all dependencies listed in the *package.json* file.

In addition to the *package.json* file, there is also the *package-lock.json* file. This file lists all installed dependencies of your application and their dependencies. It ensures that each installation of your application is similar to all other installations. In *package-lock.json*, you can find the package names, their locations in the NPM registry, and an integrity hash that protects against tampering with the package. You should version both the *package.json* and the *package-lock.json* files of your project.

You can install, update, and remove packages from your system using NPM. The following list introduces the most important commands:

- **npm install <package name>**
Downloads the specified package from the NPM repository and installs it. The package is automatically entered as a dependency in the *package.json* file of your project. You can use the `-save-dev` option to specify that this is a dependency which is only needed during development.
- **npm update <package name>**
Updates the specified package within the allowed version range specified in the *package.json* file.
- **npm remove <package name>**
Removes the package.
- **npm list**
Lists the currently installed packages.
- **npm init**
Creates a *package.json* file.

Note

Throughout the rest of this book, I will use NPM as a package manager. However, all examples can be reproduced with minor adjustments using *Yarn* or any other package manager, such as *pnpm*.

Do Not Use the Global Installation via NPM

A previously very common way to use Create React App was to install the tool globally via the `-g` option, which you can see in the following command:

```
npm install -g create-react-app
```

Listing 2.11 Installing Create React App Globally Using NPM

The global installation ensures that the tool is available to you system-wide on the command line. The drawback is that you then have only one fixed version of Create React App in your entire system and you have to take care of updating the tool yourself. Because you only need Create React App once for an application—namely, during the initialization—the risk of the tool becoming obsolete is very high. Today, Create React App even prohibits a global installation. If you try to use the tool in this mode, you'll get the following error message: "We no longer support global installation of Create React App." In this case, you have no choice but to uninstall the global installation and use one of the variants described previously.

In addition to NPM, you can use Yarn as a full-featured alternative to initialize your application.

Yarn

Yarn is a package manager for JavaScript developed by Facebook. The tool is open source and free to use.

You can install Yarn either via an installer package or via your system's package manager, depending on your operating system. For more information on the installation, you should visit <https://yarnpkg.com/getting-started/install>.

Yarn is largely API-compatible with NPM and has approximately the same feature set. At this point, you may legitimately wonder why you should bother with another package manager besides the established NPM. The reason can be found in the history of Yarn's development. If you look at the website for Yarn at <https://yarnpkg.com>, the three words *fast*, *reliable*, and *secure* will catch your eye. These are the three main areas where the earlier versions of NPM were rather weak. To address these issues, some Facebook developers created the Yarn package manager, with the following features:

■ Fast

Yarn has a caching mechanism that ensures that once installed packages do not need to be downloaded again, but can be delivered directly from the local cache. In addition, Yarn can parallelize downloads of packages, further reducing the time of installation.

■ Reliable

The *yarn.lock* file ensures that multiple installations of your application are similar, even on different systems. In older versions of NPM, only the versions of the directly installed packages were kept, but not those of their dependencies, which

sometimes resulted in serious problems. In the *yarn.lock* file, all packages to be installed, including their complete dependency trees, are specified and provided with integrity hashes.

■ Secure

The integrity hashes in the *yarn.lock* file make sure that the packages cannot be manipulated afterward. If even a small part of the package is modified, the hash value will no longer match and the installation will fail.

The statement that competition stimulates business also applies to the package manager market. After the release of Yarn, the developers of NPM quickly followed suit and fixed the biggest weaknesses in NPM, so both tools are now on par with each other.

The big advantage for you is that NPM and Yarn use the same infrastructure. If a package is available under NPM, you can also install it using Yarn and vice versa. In addition, both package managers use the *node_modules* directory to store dependencies and the *package.json* file to store the most important configurations of your application. There are only a few commands in which Yarn and NPM differ. If you install your packages via NPM using `npm install react`, you can achieve this in Yarn using the `yarn add react` command.

Initialization Using “yarn create”

Yarn is an alternative to NPM. Similar to NPM, you don't install Create React App on your system using Yarn. The following command allows you to use a temporary variant of the tool to initialize your application, after which the installation of Create React App will be discarded again:

```
yarn create react-app library
```

Listing 2.12 Initializing a React application using Yarn

No matter which variant of the installation you have chosen, after running the respective command you'll have a fully functional React application to work with. Before we get into the structure of the app in the next step, here's some more in-depth information about Create React App.

Command Line Options of Create React App

It's usually sufficient to call Create React App with only the directory name of the application you want to create. In addition, the tool provides some useful options, which we'll briefly introduce here:

■ -h, --help

This option displays a list of available options.

■ -V, --version

You can use this option to display the version of Create React App.

- **--verbose**

This option activates more extensive output. It is mainly used for troubleshooting.

- **--info**

This option is also often used in debugging. It outputs environment information such as the operating system and browser versions.

- **--use-npm**

If Yarn is installed on the system where you are running Create React App, it will be used as a package manager by default. You can use the `-use-npm` option to force the use of NPM.

- **--use-pnp**

Plug and play is a relatively new feature of Yarn. It enables you to significantly speed up the installation of dependencies of a React application because the packages are not copied to the `node_modules` directory. You can use this option to enable the plug-and-play feature for your React application.

- **--scripts-version**

The `react-scripts` package represents the core of an application that you create with Create React App. This option allows you to specify the version of `react-scripts`. Not only do you have the choice between different variants of concrete version numbers and alternative packages available through NPM, but you can also use local directories or archives.

- **--template**

The `--template` option allows you to specify a custom template for your application. Here you have several options—for example, an official Create React App template, a local directory, or an archive file. Examples of templates Create React App provides are `typescript` or `cra-template-pwa`.

Especially the newer options like `-use-pnp` or `-template` show the direction in which Create React App is evolving: from a mere tool for initializing an application to a flexible tool that offers modern development and multiple choices.

Update of an Existing Application

If you develop an application for a long time, sooner or later you will face the situation in which a new version of the `react-scripts` package is released and you should update your application to benefit from improvements. You can find out if such an update is pending by running the `npm outdated` command in your application's directory.

For example, to upgrade from version 5.0.0 to version 5.0.1, use the following command in the command line in the directory of your application:

```
npm install react-scripts@5.0.1
```

Listing 2.13 Updating the “react-scripts” Package

However, you should not perform such an update carelessly; especially with major releases, like an update of the major version number, there may be breaking changes that may limit the functionality of your application. To avoid this, you should first check the changelog at <http://s-prs.co/v570501> before each update (see [Figure 2.4](#)). There, in addition to various categories such as *Bug Fix* or *Enhancement*, you'll also find information about migrating to the new version and any breaking changes you need to be aware of.

Once you've performed the update, you must restart your application for the changes to take effect.

You must update the `react` and `react-dom` libraries separately from `react-scripts`. Again, running `npm outdated` on a regular basis will help to keep you informed about new releases. The React development team also maintains a changelog with the changes that are introduced by a new release of the library. The changelog can be found at <http://s-prs.co/v570502>.

5.0.1 (2022-04-12)

Create React App 5.0.1 is a maintenance release that improves compatibility with React 18. We've also updated our templates to use `createRoot` and relaxed our check for older versions of Create React App.

Migrating from 5.0.0 to 5.0.1

Inside any created project that has not been ejected, run:

```
npm install --save --save-exact react-scripts@5.0.1
```

or

```
yarn add --exact react-scripts@5.0.1
```

 **Bug Fix**

- `react-scripts`
 - [#12245](#) fix: webpack noise printed only if error or warning ([@Andrew47](#))
- `create-react-app`
 - [#11915](#) Warn when not using the latest version of create-react-app but do not exit ([@iansu](#))
- `react-dev-utils`
 - [#11640](#) Ensure posix compliant joins for urls in middleware ([@psiservices-justin-sullard](#))

 **Enhancement**

- `cra-template-typescript`, `cra-template`, `react-scripts`
 - [#12220](#) Update templates to use React 18 `createRoot` ([@kyletsang](#))
- `cra-template-typescript`, `cra-template`
 - [#12223](#) chore: upgrade rtl version to support react 18 ([@MatanBobi](#))
- `eslint-config-react-app`
 - [#11622](#) updated deprecated rules ([@wisammechano](#))

 **Documentation**

Figure 2.4 Excerpt from the Create React App Changelog

Automatic Update of the Source Code via “react-codemod”

Some updates to React will require changes to the existing code of the application. To avoid having to adapt every place in your application manually, you can use the `react-codemod` project. The combination of `JSCodeshift` and the `codemod` scripts allows you to have certain changes to your source code automated. Although it sounds a bit risky at first, Facebook performs it regularly on a large scale on production code.

When making changes to central interfaces, adapting the source code manually is not an option, and manipulation via regular expressions also quickly reaches its limits. For this reason, Facebook's developers implemented the `JSCodeshift` tool. It uses *recast* to transform an *abstract syntax tree* (AST) into a modified AST. These changes can include, for example, the aforementioned changes to the interfaces. The coding style is maintained as much as possible.

2.4.3 Alternatives to Create React App

Using Create React App is just one way you can start developing a React app. There are numerous other options that don't require setting up all the structures yourself right away. Popular alternatives include the following:

- **Vite**

Vite is a build tool, so it's an alternative to Webpack. It's modular and provides templates for different setups, similar to Create React App. The templates that are relevant in the context of React are `react` and `react-ts`. The `react` template builds a React application based on JavaScript. The `react-ts` template also integrates TypeScript into the development process. You can use the `npm create vite library -- --template react-ts` command to create a new React application via the `react-ts` template. For more information on the topic of setting up a React application, you should visit <https://vitejs.dev/guide/>.

- **Parcel**

Parcel also joins the ranks of web build tools. Its scope of features is similar to that of Webpack and Vite. Only the setup requires a bit more work than with the elegant template solutions in Create React App and Vite. The Parcel documentation includes step-by-step instructions for building a React application, which can be found at <https://parceljs.org/recipes/react/>.

- **Snowpack**

You can use Snowpack like the other tools for the web application build process. Compared to its competitors Parcel and Webpack, Snowpack is kept very lightweight and requires almost no configuration. Like Vite, it also provides a template system through which you can create an executable React application with just one command. On the command line, you can use the `npx create-snowpack-app react-snowpack - template @snowpack/app-template-minimal` command to set up your application. For more information, see the documentation at www.snowpack.dev/tutorials/react.

In addition, there are frameworks based on React, such as RedwoodJS or Next.js. These frameworks include their own command line tools to help you initialize and build your application.

2.4.4 React Scripts

Besides the dependencies you need for development, the `react-scripts` package provides you with another tool: *React Scripts*. This is an executable program you can use to start your application or build it for production use. The various scripts are available as entries in the `package.json` file under the `scripts` section. In total, there are four different scripts with different meanings:

- **start**

The `start` script starts the *Webpack dev server* and runs your application. In the default configuration, you can reach your application at the following URL: `http://localhost:3000`. If the port is already in use by another application, you will receive a corresponding error message on the console. You can use the `PORT` environment variable to select an alternative port.

Listing 2.14 shows how to use port 8181 as an alternative to port 3000:

```
PORT=8181 npm start
```

Listing 2.14 Running the Application on an Alternative Port

For example, in an application that you create using Create React App, you can use the ECMAScript module system with the `import` and `export` keywords. To make this work in all browsers, a combination of Babel and Webpack is used. A positive side effect of the tools used is that the application is automatically reloaded when changes are made.

It's also worth taking a look at the console from time to time, as error and warning messages are also issued here if there are any problems in the build process. In general, it's recommended to follow a zero-warning policy—that is, console output without warnings.

- **build**

During development, you should divide your application into as many small components as possible. Each of these components has exactly one purpose and is located in a separate file. The `build` script is used to pack the many component and helper files into an application bundle and optimize it to use as little memory as possible, which has a positive effect on the bandwidth required when the application is delivered to the client. If you run this script, you will find the result in the *build* directory of your application.

- **test**

The initial application is already prepared for unit testing. All necessary libraries are installed and the configuration is prepared. For the App component, the root component that Create React App generates for you by default, a simple test is already prepared as well. You can run this using the `test` script.

- **eject**

Create React App configures the app so that you can't access the configuration. However, there are situations where you may want to edit the Webpack configuration. Especially if you include additional Webpack plugins, you need to configure them. The `eject` script extracts the Webpack configuration from your application so that you can modify it yourself. But you have to be careful with the `eject` script. If you run it once, the configuration will be exported. However, this process is no longer reversible. So once you have “ejected” the configuration, you cannot bring it back.

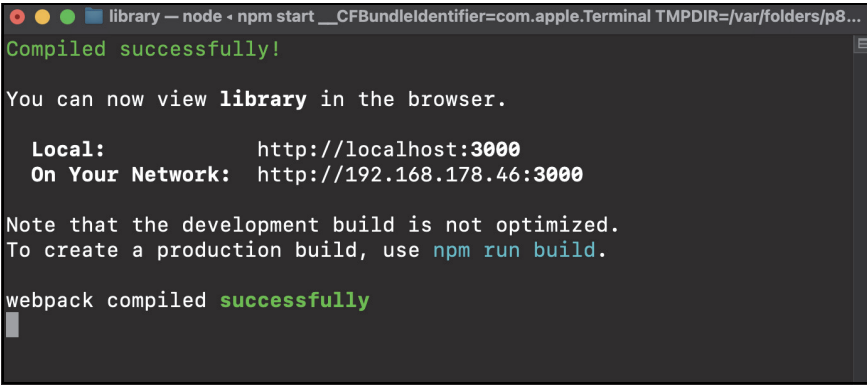
The four scripts are stored in the `package.json` file in the `scripts` section and can be executed via the package manager. For example, to start your application in development mode, you want to execute the following command in the root directory of your application:

```
npm start
```

Listing 2.15 Starting the Application in Development Mode

The script prepares the application for deployment and starts the dev server. As output, you will receive a corresponding success message (see [Figure 2.5](#)).

Running `npm start` also ensures that your application opens in your system's default browser. As a result, you'll get the view shown in [Figure 2.6](#) in the browser.



```
library — node - npm start __CFBundleIdentifier=com.apple.Terminal TMPDIR=/var/folders/p8...
Compiled successfully!

You can now view library in the browser.

  Local:            http://localhost:3000
  On Your Network:  http://192.168.178.46:3000

Note that the development build is not optimized.
To create a production build, use npm run build.

webpack compiled successfully
```

Figure 2.5 Console Output of “npm start”

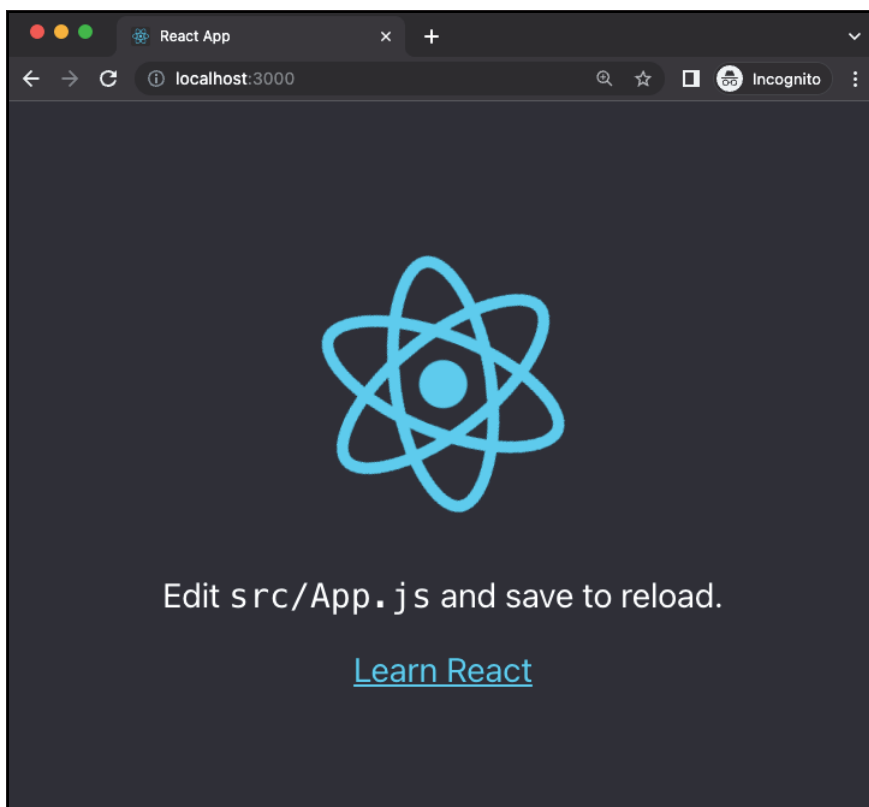


Figure 2.6 Initial React Application

The React scripts have a few more useful extensions, including server communication in development mode and encrypted communication during development. We'll cover those next.

Warning

NPM has a special feature when it deals with scripts. There are the so-called standard scripts like `start` or `test`, which you can execute directly via `npm start` and `npm test` respectively. Besides these, you can define other scripts. To run these custom scripts, you must use the `run` command. The `build` and `eject` scripts are such user-defined scripts. Thus, for a build of your application, you need to run the `npm run build` command on the command line.

2.4.5 Server Communication in Development Mode

In single-page applications, the separation between the frontend and backend is usually very strict. On the one hand, there is the React application in the browser and on the other hand, there is a server interface that can be implemented with any

programming language. During development, you can use the Webpack dev server to deliver your frontend application. You can also take advantage of convenient features such as automatic reloading when changes are made to the code. In such an application, the server-side backend takes care of data persistence and other aspects such as user authentication. The dev server does not deliver the backend. It runs as a separate server process. This means that the frontend and the backend are accessible via two different URLs. Security mechanisms in the browser prevent you from easily communicating between different servers. [Figure 2.7](#) shows the error message you get when trying to access a remote backend (*http://google.de* in this case).

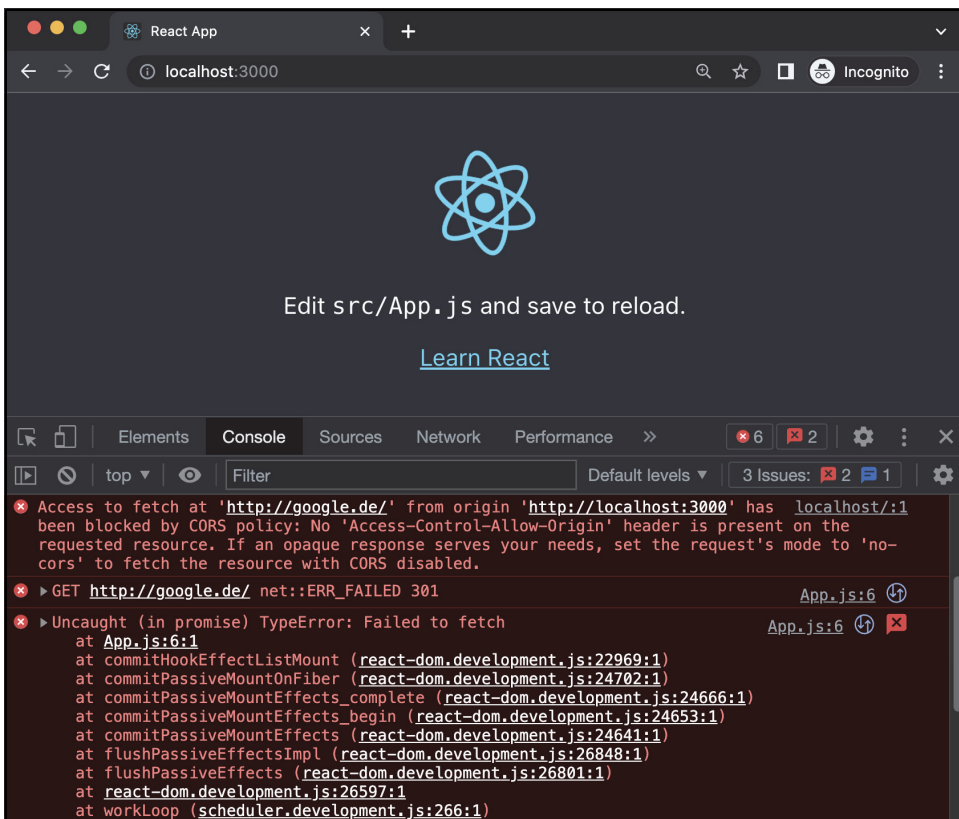


Figure 2.7 Error Message when Accessing a Backend Interface

To avoid such error messages, the Webpack dev server has a proxy extension, which ensures that all requests are sent from the frontend to the dev server, which then forwards them to the appropriate systems. All you need to do for this is to add the following entry to your *package.json* file:

```
"proxy": "http://google.de"
```

Listing 2.16 Proxy Configuration in the *package.json* File

After this adjustment, you must restart your application once for the changes to take effect. Now, as soon as you start a request to the backend, the dev server redirects it accordingly. However, for the proxy to work, the requests must be directed to the same host that delivers the application. This is usually the case in production operation, so there should be no further problems. You can learn more about server communication in [Chapter 3](#).

2.4.6 Encrypted Communication during Development

Normally, your application is delivered from the Webpack dev server over HTTP, so it is not encrypted. In most cases, this is not a problem either, as the browser interfaces that require HTTPS make an exception to the delivery of *localhost*. The use of HTTPS becomes particularly relevant if you want to use the proxy feature already presented and redirect it to an HTTPS backend. In this case, you need to set the HTTPS environment variable before starting the dev server. The server then uses a self-signed certificate to encrypt the connection. [Listing 2.17](#) shows how this works on a Unix system:

```
HTTPS=true npm start
```

Listing 2.17 Launching the Application with HTTPS Support

Note that you will then access your application via *https://localhost:3000*, so you'll have to adjust the protocol.

Now that you've launched your application and have a first impression of the development environment, we'll introduce the structure of the application in the next step.

2.5 The Structure of the Application

Create React App prepares the application for you up to the point where you can start developing right away. To help you find your way around the generated structure, this section provides a brief explanation of the various files and directories.

The base directory of your application contains several files. They contain global settings and configurations for the application:

- ***.gitignore***
This file lists the files and directories that should not be checked into the Git repository. This includes, for example, the *node_modules* directory.
- ***package.json***
In the *package.json* file, you can find the configuration of the application—for example, the start scripts and the installed dependencies.
- ***README.md***
The readme file usually contains the documentation of the application. In the initial version of this file, you'll find the description of the different start scripts and a link

to the online documentation. In your application, you should try to document in this file everything that developers need when working on the development of the application. This includes the setup of the development environment, the build and release process, and special features of the application. The goal should be that new team members only need to read the readme file and can then join the actual development process.

- ***package-lock.json***

The *package-lock.json* file stores the exact versions and integrity hashes of the installed dependencies and their subdependencies. This ensures that your application installations are the same on all systems at all times. In addition, by checking the checksums of the packages, the package manager makes sure that no manipulated packages can be injected.

Besides the files, there are also some directories in the root directory of the application:

- ***node_modules***

This directory stores all installed NPM packages in a flat hierarchy. The *node_modules* directory is reserved for the package manager, so you shouldn't manipulate its contents, lest these changes be distributed to other developers and thus lost.

- ***public***

The *public* directory contains the entry file for your application: the *index.html* file. It gets delivered upon a client request and ensures that a basic framework is available. In this directory, you can store static assets such as HTML, CSS, JavaScript, and media files. In most cases, however, it's recommended to include this data dynamically via the module system, as this gives Webpack the opportunity to optimize the content.

Placing assets in the *public* directory is recommended if the names of the resources must not be changed by the build process, as is the case with the *manifest.json* file, for example, because the browser expects the file with exactly that name. In addition, it may be useful to store files here that you do not want to be part of your application package.

Directly from within the *index.html* file, you can only reference content in the *public* directory. To reference contents of the *public* directory, you want to use the `PUBLIC_URL` variable. Within the *index.html* file, this variable is enclosed by percentage signs, as you can see in [Listing 2.18](#) for the example of the favicon:

```
<link rel="icon" href="%PUBLIC_URL%/favicon.ico">
```

Listing 2.18 Including the Favicon in index.html

To create a reference to the contents of this directory from your application (e.g., from a component), you can also use the `PUBLIC_URL` variable. This variable must get the object structure `process.env` as a prefix when accessing it, as shown in [Listing 2.19](#):

```
<img src={process.env.PUBLIC_URL + '/cat.jpeg'} />
```

Listing 2.19 Referencing Contents of the “public” Directory from the Application

■ *src*

The *src* directory is where you will spend most of your time during development. This is where you store the files that contain the components and all the helper constructs for your application.

Create React App requires a lightweight structure here, which consists of a series of files. The *index.js* file is the entry point to the file and renders the root component, which in turn is located in the *App.js* file. *App.css* contains style definitions, which are loaded in *App.js*. With *App.test.js*, you have a first unit test for your application, which you can execute via the `npm test` command. In the *index.css* file, you'll find general style definitions, such as for the *body* element of your application. Finally, the *logo.svg* file represents a static asset that is referenced from the JavaScript code of your application.

2.6 Troubleshooting in a React Application

A tool that is often underestimated in development is the web browser. You can use it not only to display your application, but also to actively integrate it into the development process. Modern web browsers have several helpful tools for analyzing and debugging an application. First and foremost, there is the debugger. You can use the shortcut `Cmd+Alt+i` on a Mac or `Ctrl+Alt+i` or `F12` on a Windows or Linux system to open the browser's developer tools. [Figure 2.8](#) shows the debugger view of a browser that has stopped at a breakpoint in the application.

To debug your application, open the developer tools, go to the **Sources** tab, and set a breakpoint by clicking on the line number. Right-clicking on the line number also gives you the option to set conditional breakpoints, where the browser stops only when a previously defined condition is met. Another way to set a breakpoint is to use the `debugger` statement. To do this, type the keyword “`debugger`” in the respective place in the source code of your application. If your browser's developer tools are open, execution will stop at this point.

You can navigate in the debugger using the toolbar in the upper-right part of the developer tools. The icons have the following meanings:

■ **Resume**

If the debugger pauses at a breakpoint, the script can be continued with this button.

■ **Step Over**

Skips the next function call.

■ **Step Into**

Jumps into the function to be called. This action adds an entry to the call stack.

■ **Step Out**

Jumps out of the current function. This action removes the current entry from the call stack.

- **Step**

This action jumps to the next line.

- **Deactivate Breakpoints**

Disables all currently set breakpoints so that execution continues without interruption.

- **Pause on Exception**

By clicking on this button, you can make sure that the execution is paused as soon as an exception is thrown.

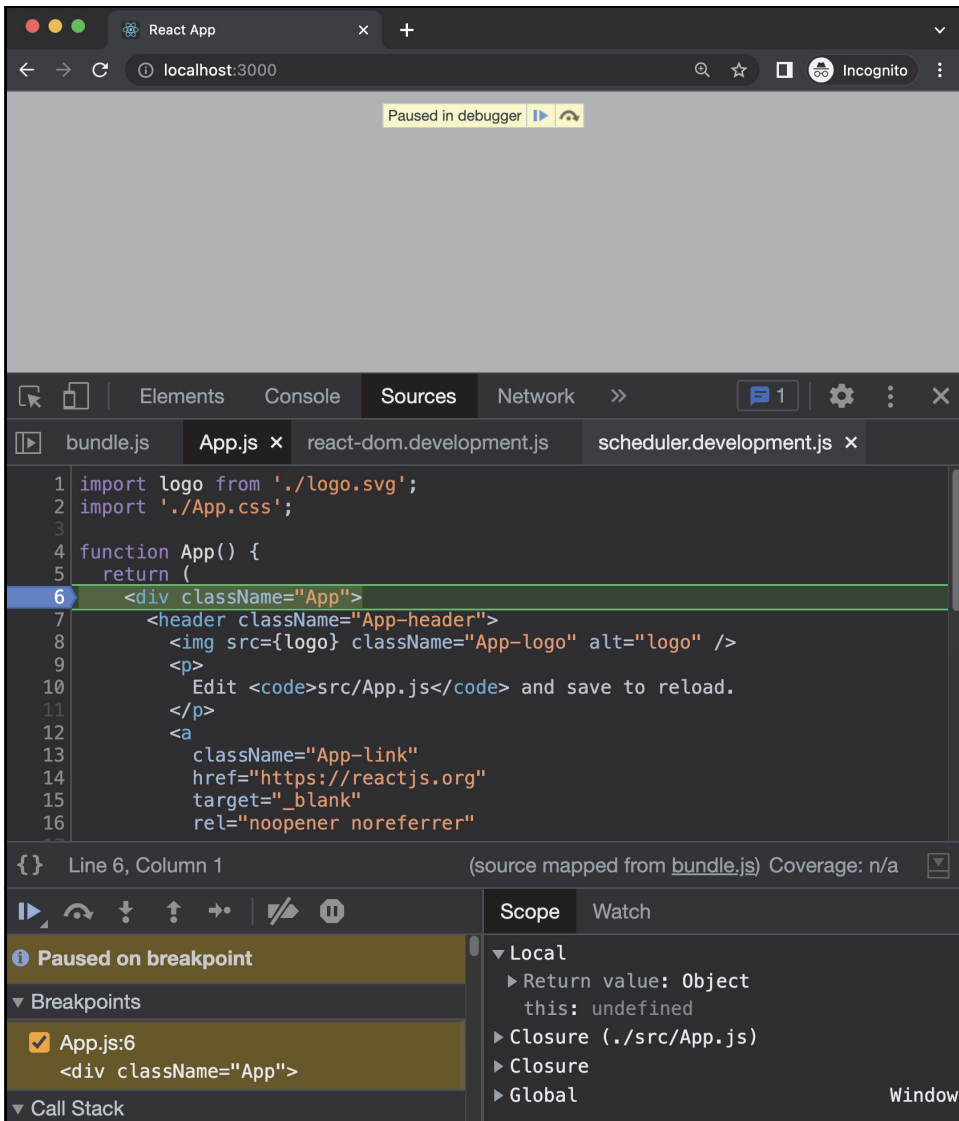


Figure 2.8 Stopped Debugger in a React Application

If the execution of your application is paused at a breakpoint, you will get additional information about the current state of your application. This way you can view the currently valid variable assignments or the call stack or define *watch expressions*, expressions that are reevaluated at each step in the debugger. The result will be displayed in the corresponding section of the debugger.

Besides using the browser debugger, there is also the option to debug your application directly in your development environment. This has the advantage that you can work directly in your source code and also set breakpoints there. For information on how to set up the debugger for your development environment, refer to the development environment's documentation. In the case of Visual Studio Code, a debugger is integrated. You can find the related documentation at <http://s-prs.co/v570509>. The documentation for WebStorm also contains a separate section on this at <http://s-prs.co/v570503>.

However, the debugger is not the only tool you can access when developing a React application. For both Firefox and Chrome, there are browser extensions that can show you the structure and dynamic data of your application. For Chrome, you can find React Developer Tools in the Chrome Web Store at <http://s-prs.co/v570504>. For Firefox, the add-on is available at <http://s-prs.co/v570505>.

After installation, you will find two tabs in your browser's developer tools labeled **Profiler** and **Components**. The **Profiler** allows you to record performance data at runtime of your application and evaluate it afterward. In the **Components** tab, you can see the current component structure of your application and inspect the individual components as well as check their props and state. [Figure 2.9](#) shows what the React dev tools look like in the browser.

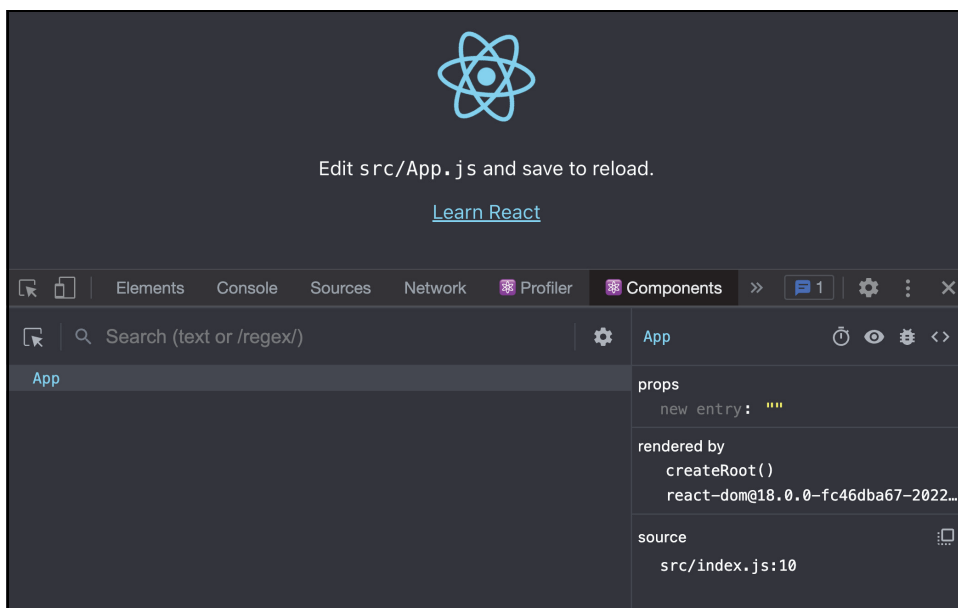


Figure 2.9 React Dev Tools in Chrome

This brings us to the final step in the development process: building the application for production use.

2.7 Building the Application

In development mode, the source code is indeed translated by the Webpack dev server so that the browser can execute it without any problem. However, the server does not optimize the code. To prepare your application for production use, you need to run the `npm run build` command.

As a result, a new directory named *build* will be created. This directory contains all the resources you need to deliver the application to your users. You can copy the contents of this directory to the document root directory of a web server to deploy your application. You can customize this behavior by adding an entry to your *package.json* file with the *homepage* key and your application URL as the value.

To test if the build worked, you can also install a local web server and deliver your application through it. One of the simplest solutions for this is provided by the `http-server` NPM package. You can run this package using the `npm run http-server build` command in the root directory of your application.

2.8 Summary

The purpose of this chapter was to introduce the React development process and the tools available to you in the process:

- You can integrate React into existing websites as well as implement a React application from scratch.
- Playgrounds like CodePen allow you to run simple experiments with React by directly including the library. However, this variant is not suitable for extensive projects.
- React can be integrated in a very lightweight way by directly including the library files in a static HTML page. However, in doing so, you forego the comfort of a prefabricated structure and configured tools.
- Create React App is the tool of choice when it comes to setting up larger applications. You can also implement extensive applications on the generated structure.
- The `react-scripts` package, which gets installed along with Create React App, provides four scripts that allow you to launch, test, build, and export the Webpack configuration.
- You can influence the behavior of Create React App as well as `react-scripts` via command line options and environment variables.

- All modern browsers have powerful developer tools to help you troubleshoot issues. In addition, the React Developer Tools extensions can help you to analyze a React application.

In the next chapter, you'll get to know the basic building blocks of React and start developing your application.

Chapter 3

Basic Principles of React

All popular single-page frameworks, such as Angular, Vue, and React, follow a similar concept when it comes to building an application: the user interface is broken down into smaller units, called *components*. React has an advantage over the other two frameworks in that components in React are very lightweight, making it easy for you to create many small and self-contained components. In this chapter, we'll look at the different aspects of components in React. You'll learn how to organize the state of your application and how the information flows through the component tree.

3.1 Preparation

If you don't have a React application at this point, you should take care of initializing one now so you can try out the concepts yourself. To do this, switch to your system's console and run the commands in [Listing 3.1](#). This way, you'll create a new application named `library`, go to the directory, and start the application.

```
npx create-react-app library
cd library
npm start
```

Listing 3.1 Creating and Launching an Application

Once you have started the application, the default browser of your system will open it automatically and display its current state with its initial component. For more information on setup and initialization, see [Chapter 2](#).

By the end of this chapter, you'll have implemented an application with a simple component hierarchy. As a template, we'll use the mockup shown in [Figure 3.1](#).

Title	Author	ISBN	Rating
			★★★★★
			★★★
			★★★★★

Figure 3.1 Application Mockup

3.1.1 Tidying Up the Application

In your application, you won't need some of the structures that Create React App has created for you, and you should always make sure that you don't have unnecessary structures in your application. So the first step is to tidy up the application. You can delete the `src/logo.svg` file. It contains the logo that will be displayed in the demo application. Likewise, you can delete the `App.test.js` file as you won't have anything to do with testing until [Chapter 9](#). You can delete the styles of the demo application in the `src/App.css` file by clearing the file.

Next, let's look at the existing structures. The two most important files in the `src` directory are `index.js` and `App.js`. A convention often used in projects is that files containing JSX structures are given a `.jsx` extension instead of `.js`. To follow this convention, you should change the file extension of both files from `.js` to `.jsx`.

3.2 Getting Started with the Application

The two most important files in a React application are the `index.jsx` and `App.jsx` files in the `src` directory. The `index.jsx` file marks the starting point of the application and is responsible for rendering. In the `App.jsx` file, you will find the root component of the application—that is, the entry point to the component tree.

3.2.1 The `index.jsx` File: Rendering the Application

The source code of the `index.jsx` file ensures that the application gets displayed.

Quick start

The `index.jsx` file is the entry point to a React application. React renders the application with a combination using the `render` method of the `root` object you create via the `createRoot` function. When you call `createRoot`, you pass the HTML element where you want React to insert the application and you pass the root component of your application to the `render` method. [Listing 3.2](#) shows a minimal version of this file, which really contains only what is most necessary:

```
import ReactDOM from 'react-dom/client';
import App from './App';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<App />);
```

Listing 3.2 Minimal Version of `index.jsx` File (`src/index.jsx`)

The source code of the `index.jsx` file Create React App generated for you is shown in [Listing 3.3](#):

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import App from './App';
import reportWebVitals from './reportWebVitals';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
);

// If you want to start measuring performance in your app, pass a function
// to log results (e.g., reportWebVitals(console.log))
// or send to an analytics endpoint. Learn more: https://bit.ly/CRA-vitals
reportWebVitals();
```

Listing 3.3 Entry Point to the Application (src/index.jsx)

The first block in the file consists of the import statements. From the React package, you use the `StrictMode` component, and you need the import of `ReactDOM` to create the root object via `createRoot`. The `index.css` file provides the styling for the current file. The `App.jsx` file exports the root component that you use as the basis for rendering the application, and importing `reportWebVitals` allows you to analyze certain metrics, which you will find explained in more detail ahead.

Import Statements Placed at the Beginning

The general structure of JavaScript files in a React application is that all required import statements are listed first. If you get the idea to distribute import statements in your files, the build process of the application prevents this with the following error message: "Import in body of module; reorder to top import/first". This is a so-called linting rule. It comes from *ESLint*, a static code analysis tool for finding bugs and anti-patterns that integrates Create React App into the build process.

You can use the `createRoot` function from the `react-dom/client` package to create the root object. When you call the function, you pass the element into which you want to render your application. The element with ID `root` is defined in the `index.html` file in the `public` directory. This is an empty `div` element, which means that your React app is initially delivered to users as a blank page, and then React builds and displays the app. The setup usually only takes a few milliseconds, so this circumstance is hardly annoying. However, there are solutions, especially with server-side rendering, to solve the problem with the initial blank page. We'll deal with the topic of server-side rendering in [Chapter 18](#).

The `root` object provides the `render` method, which you use to let React render the application. This method accepts the root component as an argument. At this point, you can see a special feature of React. If you execute this line directly in the browser, it acknowledges this with a syntax error. The notation with HTML tags directly in the JavaScript source code is called *JSX*. This is a syntax extension of JavaScript that is translated into regular JavaScript code by the build process. In this chapter, you'll learn much more about JSX and its special features. For now, all you need to know is that JSX is a convenience feature of React to keep the source code uncluttered. `<React.StrictMode>` and `<App>` enable you to reference React components.

StrictMode

The `StrictMode` component is a tool for finding potential issues in the application. It's only active in development mode and does not affect the production build.

The checks include the following:

- Finding components with unsafe lifecycle methods
- Use of the deprecated String Ref API
- Use of the deprecated `findDOMNode` method
- Finding unexpected side effects
- Use of the deprecated version of the Context API
- Recognition of insecure effects in the lifecycle

It's basically a good idea to activate these checks. An exception may be the use of older components or libraries over which you have no direct control. However, warnings and error messages here also indicate potential issues you'll have to take care of sooner or later.

In the final step, you call the `reportWebVitals` function. This step is optional.

reportWebVitals

Simply calling the `reportWebVitals` function doesn't do anything, except that a set of key figures is collected. To see the values, you must pass a callback function to the function when you call it. In the simplest case, this is a reference to `console.log`. However, you can also pass a function that sends the data to a server for further evaluation. The key figures the function provides you with are as follows:

- **Cumulative Layout Shift (CLS)**
This is the unintended movement in the layout of a page when, for example, asynchronous elements are inserted.
- **First Input Delay (FID)**
This metric denotes the time between the first user input and the browser's response to it.
- **Largest Contentful Paint (LCP)**
Specifies the rendering time of the largest image or text block within the visible area.

- **First Contentful Paint (FCP)**

This is the time between a page loading and the first content being displayed to users.

- **Time to First Byte (TTFB)**

Represents the time interval between sending a request and receiving the first byte of the response.

These figures are analyzed to assess the performance of a website. They try to include not only purely technical aspects, but also impact on users. Metrics play a big role not only for user experience, but also for search engine ranking.

3.2.2 The App.jsx File: The Root Component

The application itself currently consists of only one component, located in the *App.jsx* file. You can simplify the content of this component for the first step. The source code of the application component is shown in [Listing 3.4](#):

```
import './App.css';

function App() {
  return (
    <h1>Hello React!</h1>
  );
}

export default App;
```

Listing 3.4 Entry Component into the Application (src/App.jsx)

Once you've adjusted the source code of the application accordingly and made sure that the Webpack dev server has been started via the `npm start` command, you can view the result in the browser, which should look as shown in [Figure 3.2](#).

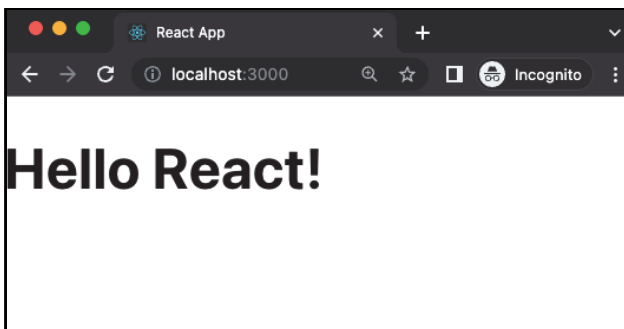


Figure 3.2 View of the Root Component of the Application

In the following section, you'll learn more about the basics of React components. You'll also implement the first component of the application and include it in the entry component.

3.3 Function Components

Quick Start

Function components are ordinary JavaScript functions that return a JSX structure. [Listing 3.5](#) shows a simple example of such a component:

```
function MyComponent() {  
  return <div>My Component works!</div>;  
}  
  
export default MyComponent;
```

Listing 3.5 Function Component

Be sure to define only one component per file. If this condition is met, you can export the component via a default export as there can be only one of this type per file. Elsewhere in your application, you can then import the component and integrate it as a tag in your JSX structure.

Components form the units that make up your application. You've already seen how to proceed with the concept in [Chapter 1](#). At this point, we want to explain the implementation of a concrete component as a functional component.

Code Style

When you develop a React application, whether as part of a team or alone, you should make sure that the source code is consistent and always follows the same formatting rules. Beginners and newcomers to development already face numerous questions about the code style, such as the naming of components or files. Numerous style guides exist to clarify these issues. One of the most popular is the *Airbnb JavaScript Style Guide*, which you can find at <https://github.com/airbnb/javascript>. In addition, there is the *JavaScript Standard Style* (<https://standardjs.com>) and the *Google JavaScript Style Guide* (<https://google.github.io/styleguide/jsguide.html>).

If you look at the descriptions of these style guides, you'll quickly realize that they are quite extensive, making it difficult to follow all the rules right away during development. ESLint is a tool that can help you in this context: it analyzes your source code and points out errors.

If you start developing your app using Create React App, ESLint will be preinstalled and will check the code both at development time and when you build the app. ESLint

provides two error levels, `warn` and `error`. Note that errors of the `error` level will cause the build to abort and you will need to fix the error before the process can finish successfully.

Create React App defines its own rule set called `react-app` and enables it by default. So you don't need to install additional packages or create a configuration for the default configuration.

But with Create React App, you aren't limited to following the given rules. You can modify the existing rules or define your very own rule set. You can do this both in a separate ESLint configuration file named `.eslintrc.json` and via an additional field named `eslintConfig` in your `package.json` file. [Listing 3.6](#) shows a sample excerpt from the `package.json` file:

```
"eslintConfig": {
  "extends": [
    "react-app",
    "react-app/jest"
  ],
  "rules": {
    "semi": "error"
  }
},
```

Listing 3.6 ESLint Configuration in the `package.json` File

The configuration extends the `react-app` and `react-app/jest` rulesets predefined by Create React App. As an additional rule, you want to define `semi`. This rule states that semicolons must not be omitted. If you still try to do so, ESLint acknowledges it with an error, and this error in turn causes the build to abort. You can test the whole thing by deleting a semicolon at the end of a line in the `App.jsx` file, for example. You can see the feedback in the console in [Listing 3.7](#):

Failed to compile.

```
src/App.jsx
  Line 6:4:  Missing semicolon  semi
```

Search for the keywords to learn more about each error.

```
ERROR in src/App.jsx
  Line 6:4:  Missing semicolon  semi
```

Search for the keywords to learn more about each error.

webpack compiled with 1 error

Listing 3.7 Error Due to a Violation of the ESLint Rules

With the appropriate ESLint plugin for your development environment, it will also show you the errors directly in the code, as shown in [Figure 3.3](#).

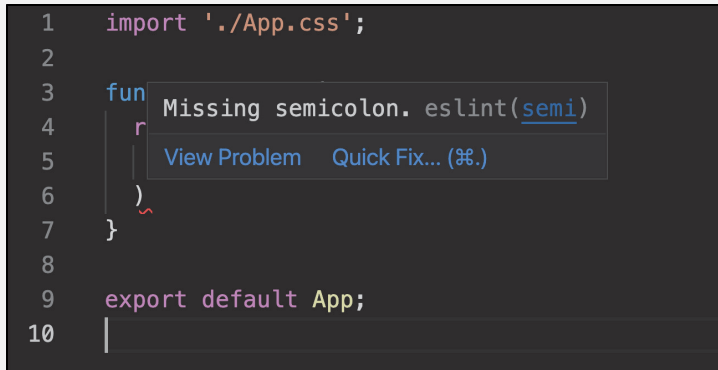


Figure 3.3 Error Message in Visual Studio Code

3.3.1 One Component per File

As a convention, it has become established in React that you create a separate file for each component. The name of this file should reflect the purpose of the component so that you can keep track of it in the file system. For example, for naming files that contain components, Airbnb's React style guide provides some guidelines:

- The file extension for a component is `.jsx` to reflect the use of JSX within the component.
- The file name is written in PascalCase, which means that the first letter is capitalized. If the name consists of more than one word, each additional word also begins with a capital letter. Aside from that, the words are not separated.
- The file name corresponds to the name of the contained component.

The component you implement in the next step represents a list of books. Consequently, a suitable name is *BooksList*. This results in the file name: *BooksList.jsx*. In a small application, you can store your components directly in the *src* directory. However, as soon as the number of files increases, you should turn to creating subdirectories. Most often in this case, the components are grouped by subject. So, for example, if your application has a user management feature, all components related to this feature will be located in the *user* directory.

But for the current example, you want to create the *BooksList.jsx* file in the *src* directory. In the first step, you develop a purely static version of this component:

```
function BooksList() {
  return (
    <table>
```

```

    <thead>
      <tr>
        <th>Title</th>
        <th>Author</th>
        <th>ISBN</th>
        <th>Rating</th>
      </tr>
    </thead>
    <tbody>
      <tr>
        <td>JavaScript–The Comprehensive Guide</td>
        <td>Philip Ackermann</td>
        <td>978-3836286299</td>
        <td>*****</td>
      </tr>
      <tr>
        <td>Clean Code</td>
        <td>Robert Martin</td>
        <td>978-0132350884</td>
        <td>****</td>
      </tr>
      <tr>
        <td>Design Patterns</td>
        <td>Erich Gamma</td>
        <td>978-0201633610</td>
        <td>*****</td>
      </tr>
    </tbody>
  </table>
);
}

```

```
export default BooksList;
```

Listing 3.8 Static Version of the “BooksList” Component (src/BooksList.jsx)

There are a few things to consider regarding the component source code. In previous versions of React, when you used JSX, which is HTML syntax in JavaScript, you had to import React so that the build process could translate the component correctly. In React version 17, the development team introduced a new JSX transform that eliminates the need for this import from React.

Import and Export

So far in this book, you have already come into contact with the ECMAScript module system. It works with the `import` and `export` keywords. Browser support for this JavaScript feature has improved significantly, but in most cases a bundler like Webpack is used in combination with a loader like Babel.

If you implement a component, you must export it so that you can include it elsewhere in your application. For these exports, you can use two variants: named and default exports. Default exports are widely used for implementing React applications.

A file can have only one default export, but multiple named exports. When importing, you can use the default import directly. You can access the named exports using curly brackets.

The `BooksList` component is a function component. This means that it consists of a JavaScript function. This is the simplest type of a React component. At this point, the Airbnb style guide recommends using named functions instead of arrow functions because you are relying on the interference of function names here. This is a feature where the JavaScript engine tries to find out the name of a function to display it in the call stack, for example.

You can think of a React component as a function that you can include, which is equivalent to a call. The result of the component is a React element. You can pass information to a component, but we'll come to that later. The structure you define and return in the component is written in JSX. Before we get into JSX in the next section, you first need to integrate the component in your application, which you can do by including it in another component.

Go to the `App.jsx` file, import the `BooksList` component, and integrate it into the structure that returns the `App` function, as shown in [Listing 3.9](#):

```
import './App.css';
import BooksList from './BooksList';

function App() {
  return (
    <div>
      <h1>Books management</h1>
      <BooksList />
    </div>
  );
}
```

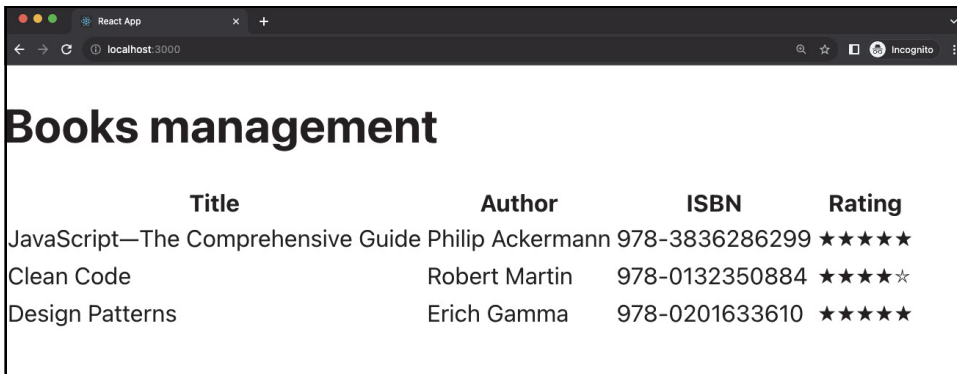
```
export default App;
```

Listing 3.9 Integration of the “BooksList” Component (src/App.jsx)

As a convention for structuring the imports in a component, it has become established that you group the imports. The first group contains imports of NPM packages; the second contains imports of custom files.

When including a component, you should note that a component must always return exactly one root element.

These customizations allow you to view the static version of your component in the browser. The result should look like the example shown in [Figure 3.4](#).



Title	Author	ISBN	Rating
JavaScript—The Comprehensive Guide	Philip Ackermann	978-3836286299	★★★★★
Clean Code	Robert Martin	978-0132350884	★★★★☆
Design Patterns	Erich Gamma	978-0201633610	★★★★★

Figure 3.4 Static Version of the “BooksList” Component

At this point, the component still doesn’t look particularly appealing. But with the help of a little CSS, you can easily fix this issue. You have already learned one method to include CSS using the *index.jsx* and *App.jsx* files as examples: you need to create a CSS file and import it into the component. So create the *BooksList.css* file in the *src* directory, and paste the contents of [Listing 3.10](#) there:

```
table {
  border-collapse: collapse;
}

th {
  border-bottom: 3px solid black;
}

tr:nth-child(2n) {
  background-color: #ddd;
}

td {
  padding: 5px 10px;
}
```

Listing 3.10 Styling of the “BooksList” Component (*src/BooksList.css*)

For the stylesheet to take effect, you want to include it as an import in the component file. Note that the style specifications in the CSS file affect the application globally.

```
import './BooksList.css';

function BooksList() {...}

export default BooksList;
```

Listing 3.11 Integrating the Stylesheet in the Component (src/BooksList.jsx)

Due to this adjustment, the book management now looks a bit more acceptable, as you can see in [Figure 3.5](#).

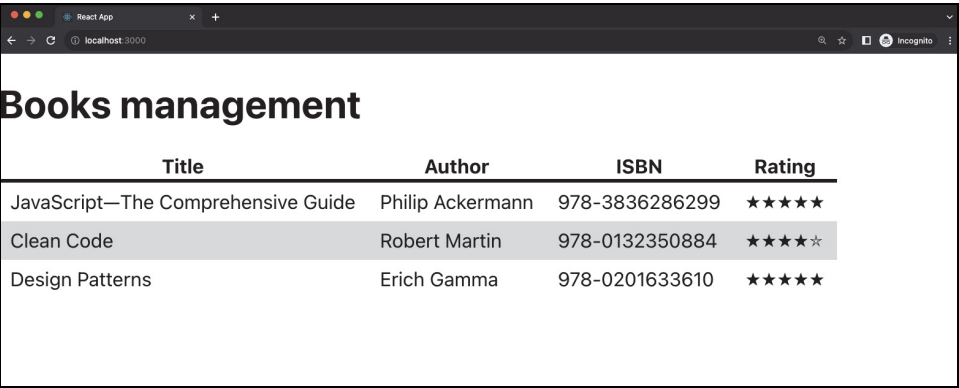


Figure 3.5 Styled View of the Books List

If you’re wondering what the JSX syntax in the components is all about, we’ll address this topic in a bit more detail in the next section.

3.4 JSX: Defining Structures in React

React introduces JSX, a language extension for JavaScript, to make it easier for you to develop components.

Quick Start

JSX is a syntax extension for JavaScript that allows you to create components and elements without much effort. JSX uses a notation similar to the tags in HTML. Within JSX, you can insert JavaScript expressions in curly brackets to display values or the result of a function call, for example.

You usually create loops by converting a data structure, predominantly an array, into a JSX structure using the `map` method.

Formulating conditions with `if` statements within JSX structures is not possible. You can either swap this out or use the ternary or the logical AND operator, for example. In [Listing 3.12](#), you can see the different aspects of JSX in use:

```
function MyComponent() {
  const name = 'React';
  const items = ['book', 'calculator'];
  const boolVal = true;

  return (
    <div>
      { /* Expression */ }
      <h1>Hello {name}</h1>
      { /* Loop */ }
      <ul>
        {items.map((item) => (
          <li>{item}</li>
        ))}
      </ul>
      { /* Conditions */ }
      {boolVal ? 'true' : 'false'}
      {boolVal && 'true'}
    </div>
  );
}

export default MyComponent;
```

Listing 3.12 Overview of the JSX Syntax

For years, web developers were taught to keep HTML, CSS, and JavaScript neatly separated. However, React breaks with this principle. There's a good reason for that: React abstracts the DOM, so you don't have to deal directly with the HTML structure. The tags you write in JSX are not ordinary HTML elements, but React elements.

React Elements

React elements are the smallest building blocks of a React application. They are simple objects that you use to describe the appearance of your application. React elements always start with a lowercase letter. An element can have attributes that are similar to the attributes of DOM elements. These are written in lowerCamelCase, such as `tabIndex`. You can enclose the values of the attributes in quotes, and then they will be interpreted as string values. For dynamic values, you use curly brackets. In this case, you can use JavaScript expressions. A peculiarity of React elements is that you are not allowed to use the `class` and `for` attributes. Because JSX is JavaScript at its core, the `class` and

for keywords are already reserved by the language itself. In React elements, you can use the `className` and `htmlFor` attributes instead.

The difference between React elements and components is that, following the convention, elements start with a lowercase letter and components start with an uppercase letter. If one of your components starts with a lowercase letter, you should either rename it directly or assign it to a variable with an uppercase letter and use that instead. In addition, an element may have one or more child elements. Elements are *immutable*, which means they are created once and cannot be modified. You can only recreate them.

When the user interface is updated, React redraws the affected regions. React optimizes rendering so that only certain parts are modified.

A component returns a structure of React elements and components, which in turn are translated into React elements. The returned structure may have only exactly *one root element*. As soon as you try to return multiple elements arranged in parallel in one component, you'll receive an error message.

JSX is a syntax extension for JavaScript, and it is compiled during the build process. Note that you don't have to use JSX to implement a React application. Alternatively, you can use the `createElement` method of React to create the building blocks of your application:

```
import React from 'react';

function MyComponent() {
  const Hello = React.createElement('span', null, 'Hello React');
  return React.createElement('div', null, Hello);
}

export default MyComponent;
```

Listing 3.13 Using `React.createElement`

The `createElement` method accepts the name of the element to be created, the inputs—also referred to as *props* of the element—and finally the child elements. This notation alternative to JSX is rarely found in React applications, as it's very inconvenient.

In JSX, you can not only model static structures, but also display dynamic content. You can insert any JavaScript expressions by enclosing them in curly brackets.

3.4.1 Expressions in JSX

In the current implementation of your component, all values are static. However, especially for more extensive content—for example, if you want to display 100 or more

books—it doesn't make much sense to build your application in this way. Before you start working on the component, you should define the data structure independently of the component. When doing this, you can choose between defining the individual books as simple objects in an array and defining a data class for a book and creating an array of instances of that class.

In React, it's common to work in a lightweight manner. Using a data class doesn't provide any advantages at this point because it doesn't contain any additional logic. For this reason, here we've used the simple array and object structure. The objects have the following properties: `id`, `title`, `author`, `isbn`, and `rating`. You can easily store the data structure in the *BooksList.jsx* file outside the component function, as shown in [Listing 3.14](#):

```
import './BooksList.css';

const books = [
  {
    id: 1,
    title: 'JavaScript-The Comprehensive Guide,',
    author: 'Philip Ackermann',
    isbn: '978-3836286299',
    rating: 5,
  },
  {
    id: 2,
    title: 'Clean Code',
    author: 'Robert Martin',
    isbn: '978-0132350884',
    rating: 2
  },
  {
    id: 3
    title: 'Design Patterns',
    author: 'Erich Gamma',
    isbn: '978-0201633610',
    rating: 5,
  },
];

function BooksList() {...}

export default BooksList;
```

Listing 3.14 Implementing the Data Structure (src/BooksList.jsx)

The data structure is only temporary, and you will later replace it with dynamic data that you obtain from a web interface. But first you need to make sure that the data is displayed in the `BooksList` component. [Listing 3.15](#) shows how to use a combination of curly brackets and the reference to `books[0].title`, for example, to display the title of the first book. In this case, you only want to display the first book; all other data records would simply be copies of this structure, which you can solve much more elegantly with a loop, thus saving yourself some error-prone typing.

```
import './BooksList.css';

const books = [...];

function BooksList() {
  return (
    <table>
      <thead>
        <tr>
          <th>Title</th>
          <th>Author</th>
          <th>ISBN</th>
          <th>Rating</th>
        </tr>
      </thead>
      <tbody>
        <tr>
          <td>{books[0].title}</td>
          <td>{books[0].author}</td>
          <td>{books[0].isbn}</td>
          <td>{books[0].rating}</td>
        </tr>
      </tbody>
    </table>
  );
}

export default BooksList;
```

Listing 3.15 Extension of the “BooksList” Component (src/BooksList.jsx)

If one of the expressions returns the `undefined` value—for example, because you mistyped it—nothing will be displayed at the corresponding position. This makes your application more stable and fault-tolerant. The same goes for the `null` or `false` values: these values won’t be displayed either. However, this fault tolerance is only sufficient to handle simple values; if you write an expression that causes an exception (such as

accessing a variable that doesn't exist), this will cause an exception in the browser, and your application will terminate.

Comments in JSX

If you need a comment in your JSX structure, either to comment out a functionality or to insert a note, you can do this using the `{/* ... */}` string, where `...` stands for your comment. So you can use an ordinary JavaScript multiline comment in a JSX expression for this purpose.

XSS Protection

Inserting dynamic content creates security risks, especially when it comes to displaying user-generated content. React has a built-in protection mechanism, which prohibits the insertion of HTML. In [Listing 3.16](#), you can see how to insert custom HTML anyway:

```
function createHTML() {
  return {
    __html: '<button onclick=\'alert("hello world");\'>hello</button>',
  };
}

function MyComponent() {
  return <div dangerouslySetInnerHTML={createHTML()} />;
}

export default MyComponent;
```

Listing 3.16 Inserting HTML

You cannot paste the HTML code directly from a variable. In that case, React would ensure that the HTML code is escaped correctly and display the HTML code directly. Instead, you need to set the `dangerouslySetInnerHTML` attribute and call a function there; in the example, that's `createHTML`. This function must return an object that has the `__html` property. The result of the sample code is a button. If you click the button, an Alert window will display, so the embedded JavaScript will be executed. Be careful when pasting HTML as it can cause serious security issues!

Currently, the list is still very tightly bound to the data structure. We'll change that in the next section.

3.4.2 Iterations: Loops in Components

In the current implementation, you display only the first record from the `books` array in the list. This severely limits the extensibility of the application. Another disadvantage

of such static structures is that you have to adjust your source code in several places as soon as you want to insert additional records, make minor changes, or insert a completely different type of record.

You can solve the list display problem by means of a loop. This means that you define the structure to be displayed only once and display it individually for each data record within the loop body. JSX doesn't provide its own feature for iteration, so instead you'll want to use JavaScript functionality. In this case, when displaying information from an array, the `map` method is the one to use. You transform the input array into an array of React elements and display them:

```
import './BooksList.css';

const books = [...];

function BooksList() {
  return (
    <table>
      <thead>
        <tr>
          <th>Title</th>
          <th>Author</th>
          <th>ISBN</th>
          <th>Rating</th>
        </tr>
      </thead>
      <tbody>
        {books.map((book) => (
          <tr key={book.id}>
            <td>{book.title}</td>
            <td>{book.author}</td>
            <td>{book.isbn}</td>
            <td>{book.rating}</td>
          </tr>
        ))}
      </tbody>
    </table>
  );
}

export default BooksList;
```

Listing 3.17 Iteration over the Book Records (src/BooksList.jsx)

The extension of the `BooksList` component is limited to the source code inside the `tbody` tag. The `books` array you defined earlier is the basis for your loop. You can use the `map` method to transform the individual records into React elements (`tr` elements in this case). Inside the callback function you pass to the `map` method, you have access to the data record in the form of the `book` object. In each iteration, you create `td` child elements for title, author, ISBN, and rating within the `tr` element. As a final extension, you add the `key` attribute to the `tr` tag. This is required by React for internal referencing.

Iterations and Keys

If you omit the `key` attribute from such an iteration, you'll receive the following warning in the browser console:

Warning: Each child in an array or iterator should have a unique "key" prop.

Listing 3.18 Iteration without the “key” Attribute

React optimizes the processing of changes by redisplaying only certain parts of the user interface. In the case of iterations, unique keys are needed for this process as they allow React to determine which elements have actually changed. The value of the `key` attribute must be unique within the iteration between elements on the same level. The assignment of the key values should not change for the elements.

The array of elements created via the `map` method is automatically displayed correctly by React. If you switch to the browser, you'll now see all three data records instead of just one.

In addition to iterations, conditions are a second important control element for designing user interfaces.

3.4.3 Conditions in JSX

You can use conditions to render certain structures depending on certain conditions. In the list display, one possible condition you can check is whether the array you want to iterate over contains any entries at all:

```
import './BooksList.css';

const books = [...];

function BooksList() {
  if (books.length === 0) {
    return <div>No books found</div>;
  } else {
    return (
      <table>...</table>
    );
  }
}
```

```

    );
  }
}

```

```
export default BooksList;
```

Listing 3.19 Conditional Rendering of JSX Structures (src/Card.jsx)

In [Listing 3.19](#), you take advantage of the fact that you can treat JSX elements like JavaScript objects and keep them as values of variables. Depending on whether the `books` array is empty or not, you can either display the information that no books were found or display the list of found books.

If, instead of an array with three elements, you assign only an empty array to the `books` variable and then switch to the browser, you'll see the text `No books found`, as shown in [Figure 3.6](#).

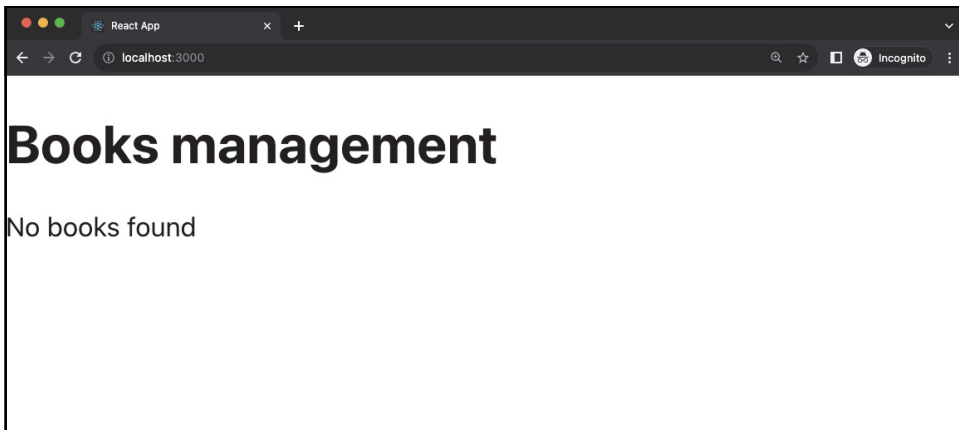


Figure 3.6 Displaying an Empty Book List

In addition to the `if` statements for conditional rendering, there are other ways to display structures under certain conditions.

Condition Shortcut 1: Logical AND operator

Instead of the `if` condition, you can use the logical `&&` operator to display elements. Whereas `if` allows you to implement multiple branches via `else if` and `else`, the logical `&&` permits only one branch. You use this branch in [Listing 3.20](#) to display a `span` element with the corresponding number of stars when a rating is present:

```
import './BooksList.css';

const books = [...];

```

```

function BooksList() {
  if (books.length === 0) {
    return <div>No books found</div>;
  } else {
    return (
      <table>
        <thead>
          <tr>
            <th>Title</th>
            <th>Author</th>
            <th>ISBN</th>
            <th>Rating</th>
          </tr>
        </thead>
        <tbody>
          {books.map((book) => (
            <tr key={book.id}>
              <td>{book.title}</td>
              <td>{book.author}</td>
              <td>{book.isbn}</td>
              <td>
                {book.rating && <span>{'*'.repeat(book.rating)}</span>}
              </td>
            </tr>
          ))}
        </tbody>
      </table>
    );
  }
}

export default BooksList;

```

Listing 3.20 Condition with the Logical AND Operator (src/BooksList.jsx)

As you can see in [Listing 3.20](#), the condition is enclosed by curly brackets. The first expression is cast to a Boolean value. This ensures that the `span` element is displayed only if the evaluation can be converted to a truth value.

Condition Shortcut 2: Ternary Operator

The disadvantage of using the AND operator is that it allows you to use only one value and no alternatives. This problem can be solved by the ternary operator. However, you should only use this operator sparingly, and never in a nested form. To disallow this

antipattern, you can enable the ESLint rule named `no-nested-ternary`. You typically use this operator less for controlling structures, and more for displaying values and their alternatives. In [Listing 3.21](#), you can see that the author is displayed if the operator is set; otherwise, the `Unknown` string will display.

```
import './BooksList.css';

const books = [...];

function BooksList() {
  if (books.length === 0) {
    return <div>No books found</div>;
  } else {
    return (
      <table>
        <thead>
          <tr>
            <th>Title</th>
            <th>Author</th>
            <th>ISBN</th>
            <th>Rating</th>
          </tr>
        </thead>
        <tbody>
          {books.map((book) => (
            <tr key={book.id}>
              <td>{book.title}</td>
              <td>{book.author ? book.author : 'Unknown'}</td>
              <td>{book.isbn}</td>
              <td>
                {book.rating} && <span>{'*'.repeat(book.rating)}</span>
              </td>
            </tr>
          ))}
        </tbody>
      </table>
    );
  }
}

export default BooksList;
```

Listing 3.21 Displaying Values with the Ternary Operator (`src/BooksList.jsx`)

3.5 Props: Information Flow in an Application

One of the most important concepts of React components is the reusability of these structures. However, if a component is static, as is currently the case with the `map` component, reusability is not possible, or at least hardly possible. For this reason, components can be parameterized. These parameters are called *props* and are passed to a component as attributes.

Quick Start

If you include a component as a tag in your JSX structure, you can define attributes. The component can access these attributes via a `props` object, which it receives as its first argument. Often the individual props are extracted directly via a destructuring statement. Listing 3.22 shows a component that accepts a prop named `person`. The component assumes that this is an object with the `displayName` property and uses this value for the output.

```
function MyComponent({ person }) {
  return <h1>Hello {person.displayName}</h1>;
}
```

```
export default MyComponent;
```

Listing 3.22 Component with Props

When you include it, you must make sure (as in Listing 3.23) that you provide the component with the correct value:

```
<MyComponent person={{displayName: 'Jane Doe'}}/>
```

Listing 3.23 Transferring Props to a Component

As you can see, you are not limited to primitive data types such as strings or numbers, but can pass any data structures such as objects, arrays, or functions to a component. Just make sure you don't forget the curly brackets here.

You can access props via the argument of the function component. Props are passed as an object whose properties you can access. In the books list, the display of individual data records is a very good candidate for a child component that receives the information for display as a prop.

3.5.1 Props and Child Components

First, you extract the block you want to swap out to the child component—that is, the `tr` element and its child elements—into a separate component named `BooksListItem` which is located in a file of the same name with the `.jsx` extension. This component needs a book data record for display, which you pass to it as a prop. You can either name

the parameter `props`, for example, and then access the information via `props.book`, or you can use the more common variant with a destructuring statement as in [Listing 3.24](#), where you insert a property of the object directly into a new variable:

```
function BooksListItem({ book }) {  
  return (  
    <tr>  
      <td>{book.title}</td>  
      <td>{book.author ? book.author : 'Unknown'}</td>  
      <td>{book.isbn}</td>  
      <td>{book.rating && <span>{'*'.repeat(book.rating)}</span>}</td>  
    </tr>  
  );  
}  
  
export default BooksListItem;
```

Listing 3.24 Child Component of the List (`src/BooksListItem.jsx`)

In the step that follows, you then include the child component in the `BooksList` component. You can see what this should look like for our example in [Listing 3.25](#):

```
import './BooksList.css';  
import BooksListItem from './BooksListItem';  
  
const books = [...];  
  
function BooksList() {  
  if (books.length === 0) {  
    return <div>No books found</div>;  
  } else {  
    return (  
      <table>  
        <thead>  
          <tr>  
            <th>Title</th>  
            <th>Author</th>  
            <th>ISBN</th>  
            <th>Rating</th>  
          </tr>  
        </thead>  
        <tbody>  
          {books.map((book) => (  
            <BooksListItem key={book.id} book={book} />  
          ))}  
        </tbody>  
      </table>  
    );  
  }  
}
```

```

        </tbody>
      </table>
    );
  }
}

export default BooksList;

```

Listing 3.25 Integration of the Child Component (src/BooksList.jsx)

The main differences from the original version of the `BooksList` component are the `import` statement you use to include the `BooksListItem` component, and its use within the component's JSX structure as a `<BooksListItem>` tag. Also in this case, the component you create in the loop needs a `key` prop.

You also pass the `book` object to the child component. Here you should make sure to not forget the curly brackets around the object, as this is a regular JavaScript object and React must interpret it as such.

Props can either be passed as a JSX string (in which case they must be enclosed in quotes), or you can pass an expression, as in our case; in the latter case, you must use the curly brackets.

As you can see, the source code of the `BooksList` component has become a bit more concise after the swap out, and this is exactly one of the goals when you integrate child components. A component should have only one purpose, and you ideally swap out all other aspects to other, specialized components.

The problem with using props is that they are JavaScript structures and are not type-safe. This means that in the example of the `BooksListItem` component, you can pass not only an appropriately shaped object via the `book` prop, but also the `true` value, for example. There are several ways to solve this problem. A very lightweight approach is to use `PropTypes`, while another option is to use `TypeScript`, which you'll learn about in [Chapter 7](#).

3.5.2 Type Safety with `PropTypes`

Since version 15.5 of React, `PropTypes` are no longer part of React itself. To use this feature, you must first install the package using the `npm install prop-types` command. After the installation, you can define the structure of the props for each of your components. You define `PropTypes` by defining a `propTypes` property on the component. The value of this property is an object. The keys are the names of the props, and the values are the respective structure. [Table 3.1](#) summarizes a selection of `PropTypes`.

PropType	Meaning
PropTypes.array	This PropTypes represents an array.
PropTypes.bigint	This type allows you to indicate that you expect a BigInt.
PropTypes.bool	This type enables you to specify that a Boolean value must be passed.
PropTypes.func	A function must be passed with this prop.
PropTypes.number	The number type requires you to pass a number.
PropTypes.object	The object type represents any object structure.
PropTypes.string	The string type is used to specify that a string is to be passed.
PropTypes.symbol	This type makes sure that a JavaScript symbol must be passed.
PropTypes.element	This type represents anything that can be rendered: numbers, strings, elements, or arrays of these types.
PropTypes.any	Here, any stands for any type.
PropTypes.instanceOf(Class)	You can use this PropTypes to specify that the prop is an instance of a particular class.
PropTypes.oneOf(['a', 'b'])	This PropTypes allows you to specify that the prop must have one of the specified values.
PropTypes.oneOfType([...])	This type works similar to oneOf, except that here you specify other types in an array.
PropTypes.arrayOf(...)	Using arrayOf, you can specify that the prop must consist of an array with certain types.

Table 3.1 Different PropTypes

These types allow you to extend your `BooksListItem` component to define the types of props to be passed:

```
import PropTypes from 'prop-types';

function BooksListItem({ book }) {...}

BooksListItem.propTypes = {
  book: PropTypes.object.isRequired,
};

export default BooksListItem;
```

Listing 3.26 PropTypes for the “BooksListItem” Component (src/BooksListItem.jsx)

[Listing 3.26](#) shows how to define `PropTypes`. After you import the `PropTypes`, you define the `propTypes` property on the `BooksListItem` function as an object with the `book` property. This is of type `PropTypes.object`—that is, an object structure. The prop is mandatory for the component to work, so you need to add another `isRequired`. If you omit this property, the props will be optional. You shouldn't use the `object`, `array`, and any general types in favor of more specific types.

You'll learn how to specify the data structure in a more detailed manner in the next section. But before that, let's look at the effects of `PropTypes` if you omit the `book` prop completely, for example. In this case, you'll receive a warning in the browser in the developer tools. However, the build process runs through without errors even if the `PropTypes` definitions are violated. [Figure 3.7](#) shows the error message.

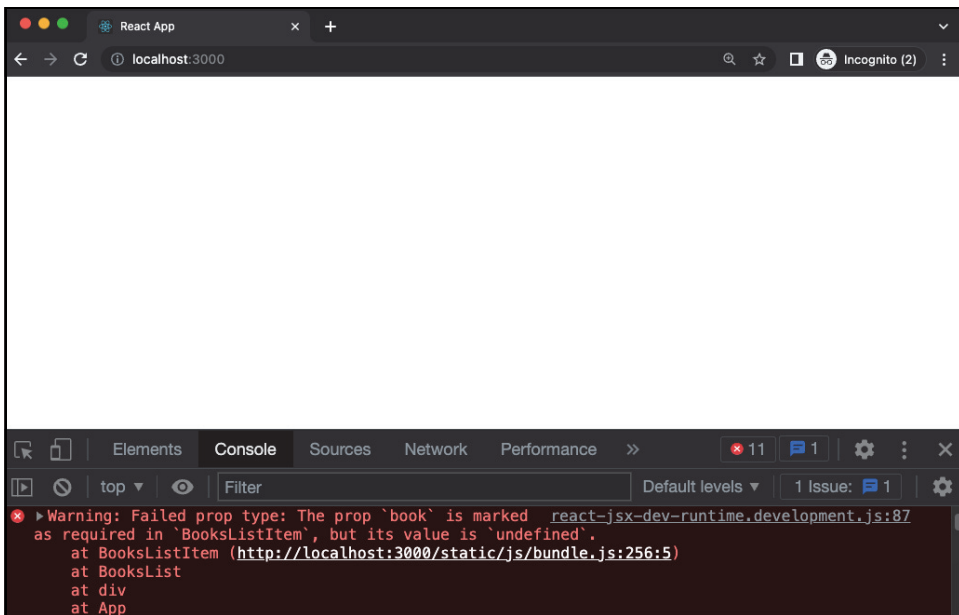


Figure 3.7 Error Message in Case of a `PropTypes` Violation

Defining Any Structures with Shapes

If the options for defining `PropTypes` presented so far are not sufficient for you, you can use shapes to influence the types in even more detail. To the shape type, you pass an object that defines the structure and can use the different `PropTypes`. [Listing 3.27](#) shows an example of how to recreate the `Book` type with `PropTypes`:

```
BooksListItem.propTypes = {
  book: PropTypes.shape({
    id: PropTypes.number.isRequired,
    title: PropTypes.string.isRequired,
    isbn: PropTypes.string.isRequired,
```

```
    rating: PropTypes.number.isRequired,  
  }).isRequired,  
};
```

Listing 3.27 “shape” PropTypes (src/BooksListItem.jsx)

PropTypes use ordinary JavaScript objects. This means that you can, for example, swap out the `Book` type to its own variable if you want to use it in multiple places. In this way, you achieve a good deal of reusability of such structures.

Until now, your components have been static and depend on a fixed data structure. However, this changes in the following step when you learn how to define the local state in a component.

3.6 Local State

Quick Start

The *state* of a component indicates the status of the component. If the state changes, React renders the component again. You can create the state via the `useState` function of the Hooks API:

```
const [state, setState] = useState(initialState);
```

Listing 3.28 Syntax of the State Hook

When you call the `useState` function, you pass the initial structure of the state. The return value of the function is an array whose first element you can use for read access to the state. The second element, a function, enables you to manipulate the state.

You can define multiple states within a function component that work independently of each other.

If you update multiple states at once, React combines these operations and renders the component only once.

In the current implementation, the data of the `BooksList` component resides in an array outside the component. The display works in this way, but this type of component state has a crucial problem: if you modify one of the objects in the array, such as a rating, the change won't be displayed. The reason is that React doesn't know about the data structure and changes. This design decision is mainly for performance reasons: if you were to define the state using simple objects, React would have to monitor them. With the introduction of the Hooks API, functional components in React have been provided with a way to manage their own state. Before the integration of this feature, a separate state was reserved for class components only.

The advantage of the state hook is that you can use multiple specialized state fragments in your component and name them accordingly. React optimizes the state changes so that they only result in a redesign of the component. So, for example, if you have three state parts in your component and update all three, this will result in the component being rendered once.

```
import { useState } from 'react';
import './BooksList.css';
import BookListItem from './BookListItem';

const initialBooks = [...];

function BooksList() {
  const [books, setBooks] = useState(initialBooks);

  if (books.length === 0) {
    return <div>No books found</div>;
  } else {
    return (
      <table>
        <thead>
          <tr>
            <th>Title</th>
            <th>Author</th>
            <th>ISBN</th>
            <th>Rating</th>
          </tr>
        </thead>
        <tbody>
          {books.map((book) => (
            <BookListItem key={book.id} book={book} />
          ))}
        </tbody>
      </table>
    );
  }
}

export default BooksList;
```

Listing 3.29 Using the “useState” Function in the Component (src/BooksList.jsx)

The customization of the existing `BooksList` component consists of first importing the `useState` function and renaming the `books` variable to `initialBooks`. The data structure remains unchanged and represents the initial value of the component state. In the

component, you call the `useState` function and pass it this value. As a return value, you get an array with two elements. At this point, you use a destructuring statement to assign the two elements to separate variables named `books` and `setBooks`.

The first element enables read access to the state. Because this variable has the same name as the original variable, you do not need to adjust anything in the component itself. You can modify the state via the second element, a function you have named `setBooks` here. You can change the data structure of the first element as well, but this does not result in any update in the rendering because React doesn't register these changes. The only way to change the state and the display in the browser is to use the setter function.

3.7 Event Binding: Responding to User Interactions

Quick Start

You can use the `on<Eventname>` props to respond to events. React defines such props for all browser events.

```
<button onClick={(event) => console.log(event)}>click me</button>
```

Listing 3.30 Click Handler in the JSX Code of a Component

When the button element is clicked, React executes the specified function and passes an object representation of the event to the function. For example, because this function is executed in the context of the component, you can adjust the state within this callback function using the setter function.

Besides `onClick`, there are numerous other event handler props, such as `onMouseOver` or `onSelect`.

Until now, you've presented a static application to your users that is capable of displaying dynamic content and that contains reusable components; however, no interaction is possible yet. But in single-page applications, such as the ones you typically implement using React, a user interaction with the application is one of the key elements. The most obvious interaction option for the books list is to adjust the rating of each data record.

3.7.1 Responding to Events

React provides you with several props that you can use to respond to events. The most commonly used variant is probably `onClick`, which allows you to respond to user clicks on specific elements.

In the books list, you allow your users to rate the books. Because the rating is a separate feature, it makes sense to swap out this functionality to a separate subcomponent. For

this purpose, you want to define a new component named `Rating` and create a new file named `Rating.jsx` in the `src` directory for it. The component receives two props. The first is a data record, which must have at least an `id` and a `rating`. Also, the component expects an `onRate` prop, which is a callback function that the component executes when a star is clicked. Second, the component calls the function with the `id` of the data record and the new `rating` value. With this structure, you have created a rating component that is largely independent of your book data structure, so you can reuse it elsewhere in your application or even in another application.

Before we get to implementing the rating component, you need to use the `npm install @mui/icons-material @emotion/styled` command to install a set of icons that you can use in your application. This will make the display of the rating component more visually appealing.

```
import PropTypes from 'prop-types';
import { StarBorder, Star } from '@mui/icons-material';
import './Rating.css';

function Rating({ item, onRate }) {
  const ratings = [];
  for (let i = 1; i <= 5; i++) {
    ratings.push(
      <button
        onClick={() => onRate(item.id, i)}
        className="ratingButton"
        key={i}
      >
        {item.rating < i ? <StarBorder /> : <Star />}
      </button>
    );
  }
  return ratings;
}

Rating.propTypes = {
  item: PropTypes.shape({
    id: PropTypes.number.isRequired,
    rating: PropTypes.number.isRequired,
  }).isRequired,
  onRate: PropTypes.func.isRequired,
};

export default Rating;
```

Listing 3.31 Implementation of the Rating Component (`src/Rating.jsx`)

Material UI

Material UI is a collection of components for React that implements Google's Material Design. It provides a variety of basic components, such as buttons, text fields, or drop-down fields. However, there are also more extensive components, such as menus or dialogs. What they all have in common is that they're available to you as components that you can import and integrate into your application. You usually influence the behavior of the components via props that you pass to the components. You can learn more about component libraries and Material UI in [Chapter 11](#).

The rating component receives as props the item—in this case, a book data record and a callback function that allows the parent component to handle a new rating. The component itself consists of an array of elements that you fill via a loop and then return. Here you can see an exception to the rule that a component must return exactly one root element: you can also return an array of elements. However, make sure here that each element has a key prop. The elements you create here are ordinary `button` elements whose `click` handlers call the `onRate` function with the `id` of the record and the rating value. In the display, you decide whether it should be a filled or empty star based on the rating value of the data record and the position in the array.

If you were to display the component this way, the stars would be surrounded by the browser's default style for buttons. You can solve this problem by defining a stylesheet for the component, saving it in a file named *Rating.css*, and importing that file into the component. However, because the styles apply to the entire application, you define a `ratingButton` CSS class for the buttons, which you assign using the `className` prop. In the stylesheet, you can then use the class selector to remove the background and border for just those buttons. You can see the corresponding stylesheet in [Listing 3.32](#):

```
.ratingButton {  
  border: none;  
  background: none;  
}
```

Listing 3.32 Styling for the Rating Component (src/Rating.css)

The next step is to save the change to the rating in the state of the parent component. For this purpose, the first step is to customize the `BooksListItem` component as shown in [Listing 3.33](#) by passing the `onRate` function from the `BooksList` component to the `Rating` component:

```
import PropTypes from 'prop-types';  
import Rating from './Rating';  
  
function BooksListItem({ book, onRate }) {  
  return (  

```

```

    <tr>
      <td>{book.title}</td>
      <td>{book.author ? book.author : 'Unknown'}</td>
      <td>{book.isbn}</td>
      <td>
        <Rating item={book} onRate={onRate} />
      </td>
    </tr>
  );
}

```

```

BooksListItem.propTypes = {
  book: PropTypes.shape({
    id: PropTypes.number.isRequired,
    title: PropTypes.string.isRequired,
    isbn: PropTypes.string.isRequired,
    rating: PropTypes.number.isRequired,
  }).isRequired,
  onRate: PropTypes.func.isRequired,
};

```

```
export default BooksListItem;
```

Listing 3.33 Rating Functionality in the “BooksListItem” Component (src/BooksListItem.jsx)

The BooksList component is responsible for the books and manages the state, which means that this is also the right place to modify the data records. So here you make sure that the rating of a book is adjusted as soon as the users adjust the rating via the Rating component:

```

import { useState } from 'react';
import './BooksList.css';
import BooksListItem from './BooksListItem';

const initialBooks = [...];

function BooksList() {
  const [books, setBooks] = useState(initialBooks);

  function handleRate(id, rating) {
    setBooks((prevState) => {
      return prevState.map((book) => {
        if (book.id === id) {
          book.rating = rating;
        }
      })
    })
  }
}

```

```
        return book;
      });
    });
  }

  if (books.length === 0) {
    return <div>No books found</div>;
  } else {
    return (
      <table>
        <thead>
          <tr>
            <th>Title</th>
            <th>Author</th>
            <th>ISBN</th>
            <th>Rating</th>
          </tr>
        </thead>
        <tbody>
          {books.map((book) => (
            <BooksListItem key={book.id} book={book} onRate={handleRate} />
          ))}
        </tbody>
      </table>
    );
  }
}

export default BooksList;
```

Listing 3.34 Modifying the Component State for an Interaction (src/BooksList.jsx)

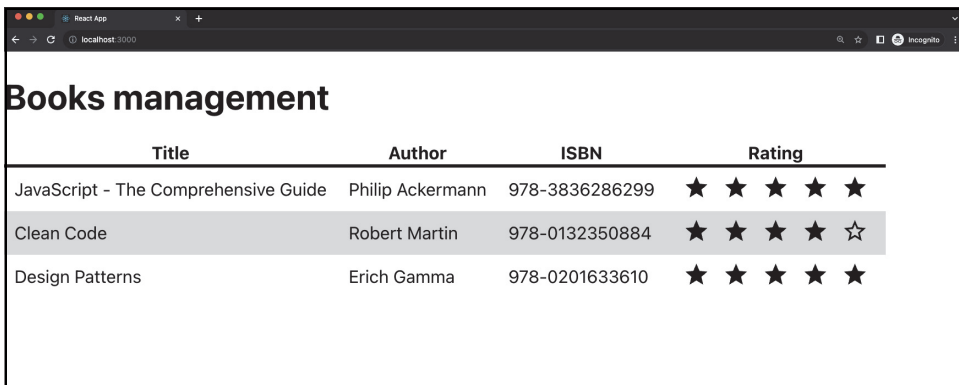
In [Listing 3.34](#), you can see that you’re allowed to define and use any functions within a component. In this context, however, you should note that these functions are recreated with every single rendering process. In this example, that isn’t a problem because it’s only one function in one central component. In [Chapter 6](#), you’ll learn about `useCallback` and `useMemo`, features of React that allow you to avoid this creation of functions under certain conditions.

You pass the `handleRate` function to the `BooksListItem` component in the `onRate` prop. The naming of the handler function and the prop follows a common naming convention that clarifies that the function is a handler function and that the prop accepts a callback function that is called on a specific event.

In the handler function, you use the state's setter function, `setBooks`, to manipulate the state. You can call this function in two ways: you can either pass a new state object that overwrites the state, or you can pass a callback function, as in this example. In this callback function, you have access to the current version of the state via the argument and create a new version on this basis, which you then return. The variant with the callback function is the safe variant when it comes to asynchronous operations: here you can be sure that you will never lose information from the state because you always work on the current version.

In the rating example, you use the `map` method of the array that is in the state and set the new `rating` value on the record that is to be modified. With these changes, you can now switch to the browser and change the ratings of the three existing data records. The result should look like the example shown in [Figure 3.8](#).

There's another variant that allows you to implement the rating of your records. We'll come to it in the next section.



Title	Author	ISBN	Rating
JavaScript - The Comprehensive Guide	Philip Ackermann	978-3836286299	★ ★ ★ ★ ★
Clean Code	Robert Martin	978-0132350884	★ ★ ★ ★ ☆
Design Patterns	Erich Gamma	978-0201633610	★ ★ ★ ★ ★

Figure 3.8 Books List with Rating Option

3.7.2 Using Event Objects

The event handlers in React allow you to access an object that represents the event in question. However, this is not the native browser implementation, but a lightweight wrapper that eliminates the idiosyncrasies of the different browsers. With version 17 of React, the team has removed one optimization in event objects: event pooling.

Event Pooling

React has long had an optimization called *event pooling*. It provided that the event objects were not created completely from scratch in each event handler, but that a manageable pool of objects was used. This optimization was supposed to lead to noticeable performance improvements, but that wasn't the case. In addition, event pooling caused confusion for developers because accessing the event object within the

handler function wasn't always possible. This is especially true for asynchronous operations, such as when there is a response from a web interface and you need information from the event object for a comparison, for example. The reason for this is that React reset the event object after the initial run of the handler function. With React 17, this feature has been completely removed. Only the so-called synthetic event has remained, but it's available to you at any time in the handler.

For the implementation of the access to the event object, you can use the example of the book rating. In the `Rating` component, you can remove the event handling completely and add a `data-value` attribute with the respective index to the `button` element. Also remember to remove the `prop` and `PropTypes` structure as well when doing such conversions:

```
import PropTypes from 'prop-types';
import { StarBorder, Star } from '@mui/icons-material';
import './Rating.css';

function Rating({ item }) {
  const ratings = [];
  for (let i = 1; i <= 5; i++) {
    ratings.push(
      <button className="ratingButton" data-value={i} key={i}>
        {item.rating < i ? <StarBorder /> : <Star />}
      </button>
    );
  }
  return ratings;
}

Rating.propTypes = {
  item: PropTypes.shape({
    id: PropTypes.number.isRequired,
    rating: PropTypes.number.isRequired,
  }).isRequired,
};

export default Rating;
```

Listing 3.35 Moving Event Handling out of the Rating Component (src/Rating.jsx)

You move the logic for event handling from the `Rating` component to the parent `BooksListItem` component. In [Listing 3.36](#), you can see the customized source code of the `BooksListItem` component:

```

import PropTypes from 'prop-types';
import Rating from './Rating';

function BooksListItem({ book, onRate }) {
  function handleRate(event) {
    const rating = event.target.closest('[data-value]')?.dataset.value;
    if (rating) {
      onRate(book.id, parseInt(rating, 10));
    }
  }

  return (
    <tr>
      <td>{book.title}</td>
      <td>{book.author ? book.author : 'Unknown'}</td>
      <td>{book.isbn}</td>
      <td onClick={handleRate}>
        <Rating item={book} />
      </td>
    </tr>
  );
}

BooksListItem.propTypes = {...};

export default BooksListItem;

```

Listing 3.36 Event Handling with Access to the Event Object (src/BooksListItem.jsx)

In the `BooksListItem` component, you first define the function that will handle the event. The parameter list consists of the reference to the event object. This event object provides access to the `target` property, which contains the element that was clicked. The rating value is in the `data-value` attribute. However, you don't know on which element exactly the click event was triggered: whether on the `svg` element of the icon, the button, or the table cell.

You can solve this problem using the `closest` method. It finds the closest element to which the passed selector applies. When you use this call, it can happen that no element is found. This is the case when you click on the `td` element. If that happens, you cannot access `dataset.value` and your browser throws an exception. You can use the optional-chaining operator (`?.`) to intercept this problem and assign the `undefined` value to the `rating` variable instead. After that, you need to check if `rating` has a valid value, and then you can call the `onRate` function that was passed via the props. Note that the value of `data-value` is interpreted as a string. So if you need a number, you have to take care of the conversion—using the `parseInt` function, for example.

This conversion saves quite a few callback functions by dragging the event handler in the tree closer to the component responsible for the state. Whether you choose this option or the previous one is up to you. The only important thing is that you remain as consistent as possible in your implementation and solve similar problems with the same approach. This is important because there are usually multiple solution options in React, and you will improve the readability and understandability of your code.

3.8 Immutability

If you modify the state of a React component, the display in the browser may not update. This has to do with the fact that you have modified a deep object structure, but React detects a change only at the top level. For this reason, in the `BooksList` component, you also used the `map` method in the `handleRate` function, which doesn't modify the original array, but works on a copy. You can reproduce the issue by slightly adjusting the `handleRating` function in the `BooksList` component. [Listing 3.37](#) shows the new structure. For comparison, we've also included the original variant in a comment.

```
function handleRate(id, rating) {
  setBooks((prevState) => {
    const index = prevState.findIndex((book) => book.id === id);
    prevState[index].rating = rating;
    return prevState;

    /* original implementation
    return prevState.map((book) => {
      if (book.id === id) {
        book.rating = rating;
      }
      return book;
    });
    */

  });
}
```

Listing 3.37 Direct Modification of the State that Does Not Cause Any Re-rendering
(src/BooksList.jsx)

In this customized variant, you search for the index of the record you want to modify. You then access the state structure via this index, change the value, and return the state object. If you set a breakpoint in the code before the return, you will see in the debugger that the `rating` value has been set correctly. The problem here is that React does not recognize the change.

But even the solution with the `map` method is not perfect. React does recognize the change because you are returning a copy of the state object and not the original. But during the operation, you also modified the original. The reason is that you iterate over objects and you manipulate these objects, which also affects the entries of the original array.

In general, you should always be careful with nested object structures. You'll often find expressions like `{...myObj}` for objects or `[...myArr]` to copy the corresponding structures. However, these operations only create a flat copy; that is, they only copy the top object or the top array, not references at lower levels.

If you are aware of this, it isn't a problem in most cases. But such operations can lead to unwanted side effects. For this reason, you should always use so-called immutable data structures in such cases. This means that you do not modify the original data structure, but always create a deep copy of it. Creating such a deep copy can require some effort. For this reason, there are several libraries that deal with this. Examples include *immunity-helper*, *Immer*, or *Immutable.js*. The libraries take different approaches and range from a simple solution like *immunity-helper* that supports a function and a defined set of operations, to a flexible library like *Immer* that uses JavaScript proxies, to comprehensive solutions that provide custom data types for immutability. In the following sections, I'll show you *Immer* as a possible solution in use in a React application.

3.8.1 Immer in a React Application

Immer is an independent library that you can use both in the frontend and in Node.js in the backend. Before you can use *Immer*, you must install it using the `npm install immer` command. The core of *Immer* is the `produce` function. You pass the object to it on which you want to perform an operation and a callback function. The return value of `produce` is the modified copy of the object. In the callback function, you have access to the object and can work with it. For example, you can assign a new value to a property, and *Immer* makes sure that this operation doesn't change the original object.

In [Listing 3.38](#), you use the `produce` function of *Immer* to repair the code from the previous example, which modified the state directly and did not change the display.

```
import { useState } from 'react';
import produce from 'immer';
import './BooksList.css';
import BookListItem from './BookListItem';

const initialBooks = [...];

function BooksList() {
  const [books, setBooks] = useState(initialBooks);
```

```
function handleRate(id, rating) {
  setBooks((prevState) => {
    return produce(prevState, (draftState) => {
      const index = draftState.findIndex((book) => book.id === id);
      draftState[index].rating = rating;
    });
  });
}

if (books.length === 0) {
  return <div>No books found</div>;
} else {
  return (
    <table>
      <thead>
        <tr>
          <th>Title</th>
          <th>Author</th>
          <th>ISBN</th>
          <th>Rating</th>
        </tr>
      </thead>
      <tbody>
        {books.map((book) => (
          <BooksListItem key={book.id} book={book} onRate={handleRate} />
        ))}
      </tbody>
    </table>
  );
}
}

export default BooksList;
```

Listing 3.38 Using Immer in the “BooksList” Component (src/BooksList.jsx)

The `handleRate` function remains almost unchanged, except for one small modification. You pass the current state, which is in the `prevState` variable, to the `produce` function. In the callback function, you look for the index of the record in the `draftState` variable and then change the `rating` value. Again, you can use the debugger and breakpoints and you will notice that the original state is not changed. You can use Immer in such error-prone situations or whenever you manipulate the state. In [Listing 3.39](#), you can see the `handleRate` function in the original version in combination with Immer:

```
function handleRate(id, rating) {
  setBooks((prevState) => {
    return produce(prevState, (draftState) => {
      draftState.map((book) => {
        if (book.id === id) {
          book.rating = rating;
        }
        return book;
      });
    });
  });
}
```

Listing 3.39 Using Immer in the Original “handleRate” Function (src/BooksList.jsx)

You can see in this example that the source code of the function gets a bit more extensive and also slightly more difficult to read. However, the advantage of this implementation is that you can exclude the modification of the original structure.

3.9 Summary

In this chapter, you got to know the basic principles of React:

- Components form the building blocks of a React application.
- Function components represent the more modern, lightweight, and flexible variant of components.
- A function component returns a JSX structure that takes care of rendering in the browser.
- Conditions allow you to display certain parts of the structure depending on whether a certain condition is true. You can implement conditions with `if` statements, the ternary operator (`?:`), or even a logical AND.
- Because JSX is a syntax extension of JavaScript, you can implement loops using the `map` method of arrays, for example.
- Props let you pass information to a component via its attributes.
- `PropTypes` allow you to secure the types of props of a component.
- You can register event handlers with attributes like `onClick`.
- You can map the state of a component via the state hook using the `useState` function. The first element of the return value allows you to read the state; the second element allows you to modify the state.
- You pass the initial state to the `useState` function when you call it.

- If you pass a function to the setter function of the state hook, you can ensure that you are using the state that is valid at the time of the call.
- React relies heavily on immutable data structures for change detection. For this purpose, you can either manually create copies of data structures and execute the changes on them, or use established libraries such as *immutability-helper*, *Immer*, or *Immutable.js*.

In the next chapter, you'll learn about other aspects of components, such as their life-cycle, and you'll learn how to communicate with a server interface.

Chapter 4

A Look Behind the Scenes: Further Topics

With a component hierarchy and the data flow between the parent and child components with props and with the help of communication via functions that you pass from the parent component to the child component through props, numerous cases can already be covered. However, numerous other patterns have become established in React, which you can use to structure your React application in such a way that it still remains manageable even with a larger range of functions.

However, before we get into the actual patterns, you'll first learn about the lifecycle of a component.

4.1 The Lifecycle of a Component

A component in an application goes through a lifecycle that you can divide into the following three stages:

- **Mount**

The first step in the life of a component is that it is hooked into the component tree and thus rendered. During this stage of the lifecycle, you mostly perform initialization tasks.

- **Update**

Very few components are purely static. Most of the time they contain dynamic information that can change during the lifetime of the component. Such a change entails an update of the component. Those updates can occur more frequently and sometimes unintentionally.

- **Unmount**

The last stage in the lifecycle of a component is the unhooking of the component from the component tree. You have the option to execute logic at this point as well. Most often, you use this opportunity to free up resources that the component has requested.

The lifecycle of function components differs from that of class components. In this chapter, we focus on function components. You'll learn more about class components and their lifecycle in [Chapter 5](#).

4.2 The Lifecycle of a Function Component with the Effect Hook

Quick Start

The `useEffect` function allows you to intervene in the various stages of a component's lifecycle. The `useEffect` function accepts a callback function and an optional array as arguments. The syntax looks as follows:

```
useEffect(effectFunction, dependencies)
```

This function covers all three stages of the lifecycle:

■ Mount

You pass a callback function which is supposed to be executed on mount. You also pass an empty array as the second argument.

■ Update

Besides the callback function, you either pass no second argument, in which case the callback is executed on each update; or you specify an array, and then the function will be executed only if one of the specified dependencies has changed.

■ Unmount

You can respond to the unhooking of your component from the component tree by again returning a function from the callback function. This function gets executed upon the removal of the component.

So far, you only know the component function itself, but you haven't yet intervened in the lifecycle of the component, at least not consciously. The component function executes React as the first step in the lifecycle. Here, however, you should refrain from using any side effect as React is capable of interrupting the render process, resuming it later, or canceling it entirely. Examples of a side effect include operations that aren't directly related to rendering the component, such as communicating with a web server to load data or setting a timeout or interval. If you were to place such a side effect in the component function directly and the render process were to be aborted and restarted, then the side effect would be executed a second time, which in the best case does no further damage, but can also cause serious problems.

In the following sections, you'll learn about the different stages of the component lifecycle and how you can intervene there using the effect hook.

4.2.1 Mount: The Mounting of a Component

Now that you know that you aren't allowed to trigger side effects directly in the component function, you need a place where you are allowed to do so. As an example, we want to use the books list from the previous example again, but in a much simpler variant. The component manages its own state and stores the data records there as an array. Initially you start with an empty array. Once the component has been loaded, you fill the state with data. [Listing 4.1](#) shows the code of the component:

```
import { useState } from 'react';
import './BooksList.css';
import { useEffect } from 'react';

function BooksList() {
  const [books, setBooks] = useState([]);

  useEffect(() => {
    setTimeout(() => {
      setBooks([
        {
          id: 1,
          title: 'JavaScript - The Comprehensive Guide',
          author: 'Philip Ackermann',
          isbn: '978-3836286299',
          rating: 5,
        },
      ]), 2000);
    }, [books]);

    if (books.length === 0) {
      return <div>No books found</div>;
    } else {
      return (
        <table>
          <thead>
            <tr>
              <th>Title</th>
              <th>Author</th>
              <th>ISBN</th>
            </tr>
          </thead>
          <tbody>
            {books.map((book) => (
              <tr key={book.id}>
                <td>{book.title}</td>
                <td>{book.author}</td>
                <td>{book.isbn}</td>
              </tr>
            ))}
          </tbody>
        </table>
      );
    }
  });
}
```



```
}  
}  
  
export default BooksList;
```

Listing 4.1 Asynchronous Filling of the Component State during Mounting (src/BooksList.jsx)

The `BooksList` component renders the empty array from the initial state in the first step. This means you first get a view in the browser like the one shown in [Figure 4.1](#).

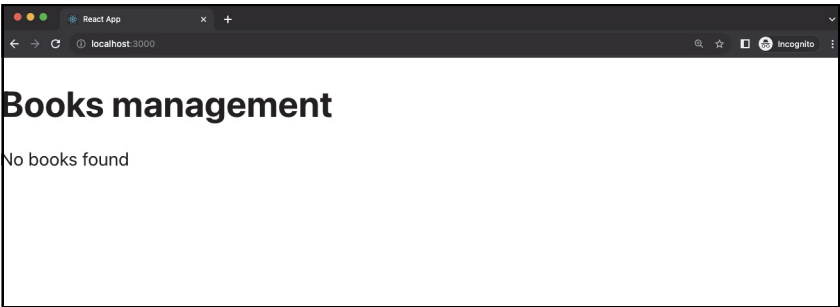


Figure 4.1 Initial Representation of the “BooksList” Component

The new aspect in the component is the call of the `useEffect` function. Note at this point that the function is called with two arguments. The first one is a callback function and the second one is an empty array. The empty array plays a significant role here. It specifies that the callback function you pass as the first argument is executed only once when the component is mounted. If you omit this array, React executes the callback function on every update. You will see what the consequences are in the next section.

Inside the callback function you pass to `useEffect`, you first set a timeout of two seconds. This is just to help you see the effect of the mount hook. In the timeout function, you overwrite the current state of the component with a new array containing a data record that is displayed afterward by your component. After these two seconds, you’ll see the final display in the browser, as shown in [Figure 4.2](#).

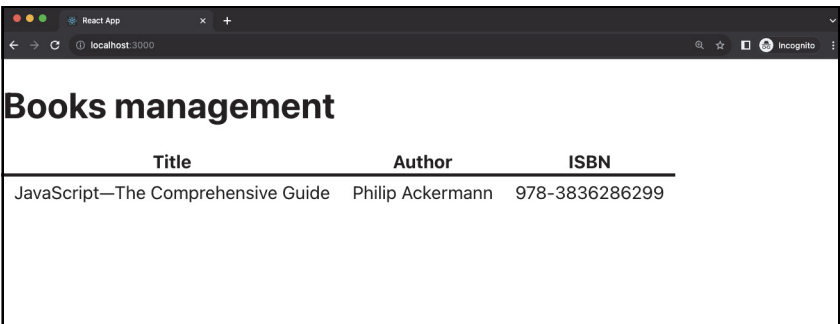


Figure 4.2 Updated Display after the Mount Hook

Usually, the time span between these two views is relatively short. But you should be aware that your users will see both variants. You can take advantage of this fact and, for example, display a loading indicator or something similar to inform your users that the data will be displayed soon. In the best case, they won't notice the loading indicator. But if the operation takes a little longer, the users will be informed that an action is still being performed in the background.

Another aspect of the effect hook when mounting the component is that the component function is executed twice in this case. The first time the component is rendered with a message that there is no data, and then the second time React displays the record. Normally, such component functions are lightweight, so this won't matter, especially if you avoid side effects. Nevertheless, you should be aware that the function is called significantly more often than just once.

4.2.2 Update: Updating the Component

As soon as the state of a component changes, React redraws it. In this case, the component enters the update stage of its lifecycle. The effect hook enables you to intervene here. This option becomes relevant, for example, when you need to react to updates of the state. State updates are asynchronous in React. So you can't attach logic in the sense of "execute this function after the state has been updated" directly to it. And this is where the effect hook comes into play. The code in [Listing 4.2](#) illustrates this:

```
import { useState } from 'react';
import './BooksList.css';
import { useEffect } from 'react';

function BooksList() {
  const [books, setBooks] = useState([]);

  useEffect(() => {
    setTimeout(() => {
      setBooks([
        {
          id: 1,
          title: 'JavaScript - The Comprehensive Guide',
          author: 'Philip Ackermann',
          isbn: '978-3836286299',
          rating: 5,
        },
      ]);
    }, 2000);
  }, []);
}
```

```
useEffect(() => {
  console.log('Elements in the state: ', books.length);
  console.log(
    'Table rows: ',
    document.querySelectorAll('tbody tr').length
  );
});

if (books.length === 0) {
  return <div>No books found</div>;
} else {
  return (
    <table>
      <thead>
        <tr>
          <th>Title</th>
          <th>Author</th>
          <th>ISBN</th>
        </tr>
      </thead>
      <tbody>
        {books.map((book) => (
          <tr key={book.id}>
            <td>{book.title}</td>
            <td>{book.author}</td>
            <td>{book.isbn}</td>
          </tr>
        ))}
      </tbody>
    </table>
  );
}
}

export default BooksList;
```

Listing 4.2 Responding to Updates in the Component (src/BooksList.jsx)

In this example, you can see that React allows you to make more than one `useEffect` call in a component. The first `useEffect` takes care of mounting the component, so it is executed exactly once. On the second call of `useEffect`, you omit the second argument. This causes React to execute the callback function on every render operation. So the output in the browser console is as shown in [Listing 4.3](#):

```

Elements in the state: 0
Table rows: 0
Elements in the state: 0
Table rows: 0
Elements in the state: 1
Table rows: 1

```

Listing 4.3 Output of the Effect Hook

The Effect function is therefore executed exactly three times:

- The first time is the initial representation. Each update function is also executed initially.
- The second output block is from `StrictMode`. In *strict mode*, React renders all components twice to detect potential issues. However, this only happens in the development process. You can disable this behavior by removing the `React.StrictMode` component in the `index.jsx` file. But doing so takes away React's ability to warn you about potential issues.
- Finally, the third output occurs after the state update that you perform in the first call of `useEffect`. As you can see, React performs the Effect function after the state and DOM updates. So you can be sure that the display is already updated as soon as the function is executed.

Warning: Avoid Direct DOM Access

Please do *not* try to interact with the DOM directly. The use of `document.querySelector` is for demonstration purposes only. In most cases, accessing the DOM indicates an antipattern in your application that you should be cautious about. React works declaratively, so you describe the target state of your application. Direct DOM operations fall into the realm of imperative programming.

Selective Updates

You can control the execution of the Effect function with the second argument, the dependency array. This is shown in the example in [Listing 4.4](#):

```

import { useState, useEffect } from 'react';

function MyComponent() {
  const [state1, setState1] = useState(0);
  const [state2, setState2] = useState(1);

  useEffect(() => {
    setTimeout(() => setState1(1), 1000);
  }, []);
}

```

```
    setTimeout(() => setState2(2), 2000);
  }, []);

  useEffect(() => {
    console.log('State1 changed: ' + state1 + ' state2: ' + state2);
  }, [state1]);

  useEffect(() => {
    console.log('State2 changed: ' + state2 + ' state1: ' + state1);
  }, [state2]);

  return <div>MyComponent works</div>;
}

export default MyComponent;
```

Listing 4.4 Demonstration of Various State Changes (src/MyComponent.jsx)

This example is somewhat contrived, but it nicely illustrates the implications of the second argument of the `useEffect` function. The component has two substates named `state1` and `state2` and three calls of the `useEffect` function. In the first `useEffect`, you set a timeout for each change of `state1` and `state2`. The second `useEffect` takes care of the change of `state1`, and the third `useEffect` is triggered when `state2` is changed.

The output you get consists of a total of six lines in the browser. The first four are from the initial mount of the component as both `useEffects` for the state changes are also executed initially. In addition, `StrictMode` takes effect here again, which provides for a doubling of the output. After one second, `state1` is changed, which triggers the `useEffect` function with `state1` as a dependency. After another second, `state2` is changed, which triggers the final `useEffect` function.

In the console where your build process is running, you'll see the following warning messages:

```
WARNING in src/MyComponent.jsx
```

```
Line 14:6:  React Hook useEffect has a missing dependency: 'state2'. Either include it or remove the dependency array  react-hooks/exhaustive-deps
```

```
Line 18:6:  React Hook useEffect has a missing dependency: 'state1'. Either include it or remove the dependency array  react-hooks/exhaustive-deps
```

Listing 4.5 Dependency Warnings in the Console

You get these warnings when React detects that you are accessing external values in your `useEffect` callback but you haven't listed them as a dependency. This means that your effect relies on external structures but doesn't automatically execute when they change, indicating a potential source of error.

4.2.3 Unmount: Tidying Up at the End of the Lifecycle

Only in rare cases do you need the unmount stage in the lifecycle of a component. Typically, this is only the case if you requested a resource when mounting the component. Examples of such resources are data streams like WebSockets, but also intervals—that is, callback functions that are executed regularly via `setInterval`. [Listing 4.6](#) shows an example that contains an interval:

```
import { useState, useEffect, useRef } from 'react';

function MyComponent() {
  const [show, setShow] = useState(true);

  useEffect(() => {
    setTimeout(() => setShow(false), 5000);
  });

  return show ? <Child /> : <div />;
}

export default MyComponent;

function Child() {
  const intervalRef = useRef(null);
  const [time, setTime] = useState(0);

  useEffect(() => {
    console.log('Mount');
    intervalRef.current = setInterval(() => {
      setTime((prevTime) => prevTime + 1);
    }, 1000);

    return () => {
      console.log('Unmount');
      clearInterval(intervalRef.current);
    };
  }, []);

  return <div>{time}</div>;
}
```

Listing 4.6 Deleting an Interval when Unmounting (src/MyComponent.jsx)

The example consists of two components. The outer component named `MyComponent` displays the child component and ensures that it's removed after five seconds. You can

do this by creating a new state with the value `true`. In an effect hook, you set a timeout in the callback function of which you change the state to the value `false` after five seconds. You use this value during rendering to decide whether to display the child component or an empty `div` element.

The child component, `Child`, also has its own local state where you store the number of seconds that have passed since mounting. In the callback function of the `useEffect` call, you first output the `mount` string on the console, then define an interval that increments the state value by one every second. Again you define a function as the return value of the callback function. Here you first output the `Unmount` string to the console and then break the interval with the `clearInterval` function.

The `useRef` function is a prequel to [Chapter 6](#), where you'll learn even more about the Hooks API of React. At this point, let's just say that the `useRef` hook can store an unchanging value over multiple render cycles. This allows you to ensure that you always have access to the same interval identifier.

When you integrate the component and switch to your browser, you'll see a number in the display that increases by 1 every second starting from 0 until the value 4 is reached. After that, the number is hidden again. If you open the console in the browser, you'll see the following output there:

Mount	MyComponent.jsx:26
Unmount	MyComponent.jsx:20
Mount	MyComponent.jsx:26
Unmount	MyComponent.jsx:20

Listing 4.7 Console Output of the Unmount Example

The double output of `mount` and `unmount` is again from `StrictMode`, where React double mounts the component as a test. If you observe the console, you will notice that the last `unmount` is output only after a delay of five seconds—that is, when the `Cleanup` function of the effect hook is executed.

If you don't implement the `unmount` logic, the interval continues to run in the background after the component is unmounted. Depending on how often you use the component in your application, this can lead to performance issues.

Note

Whenever you request a resource, you must also release it again!

Cleanup

In a way, handling the unhooking of a component is just a special case of the cleanup routine of the effect hook. The function you return in the callback function to the

`useEffect` function is executed before each new execution of the callback function to perform cleanup operations. An example of this is shown in [Listing 4.8](#). The timer component shows the number of seconds since it was mounted. You can also reset the counter via a button.

```
import { useState, useRef, useEffect } from 'react';

function Timer() {
  const [time, setTime] = useState(0);
  const [reset, setReset] = useState(null);
  const intervalRef = useRef(null);

  useEffect(() => {
    console.log('useEffect');
    setTime(0);
    intervalRef.current = setInterval(() => {
      setTime((prevTime) => prevTime + 1);
    }, 1000);
    return () => {
      console.log('clearInterval');
      clearInterval(intervalRef.current);
    };
  }, [reset]);

  return (
    <div>
      <div>{time}</div>
      <button onClick={() => setReset(Math.random())}>reset</button>
    </div>
  );
}

export default Timer;
```

Listing 4.8 Timer Component with Cleanup Routine (src/Timer.jsx)

For the component, you need two states:

- In one of them, you store the number of seconds that have passed. You display this value when you render the component.
- The second state is used to control the resetting of the timer. Because the timer again uses an interval, you need a Ref object, which you create using the `useRef` function.

The core element of the timer is the call of the `useEffect` function. In the callback function, you first create a console output and then set the timer to the value 0. Then you

start the interval that increases the `time` value by 1 every second. As a return value, you define a function that also generates a console output and cancels the interval. As a dependency of the effect hook, you set the `reset` state.

In this way, you have created an effect that will be executed every time `reset` gets updated. When mounting the component, React executes the callback function for the first time, starting the timer. By clicking on the button, you update the `reset` state with a random value. The effect callback will then be executed again. This cancels the interval and restarts the timer.

So you have now also become acquainted with an example of a cleanup routine with the effect hook. In the next section, we'll look at an operation you'll need very often in a React application: loading data from a server interface.

4.3 Server Communication

After the description of the component lifecycle, it's time to return to our running example, the books list, which we'll use to describe the server communication.

4.3.1 Server Implementation

At this point, you access a server interface in read-only mode. The server provides the data for the list and has no logic itself. For this reason, you can use a simple server implementation such as the `json-server` NPM package. You can install this package in your application using the `npm install json-server` command. In addition to the installation, you need a JSON file to store the data, which is the basis for the server. An example of such a file is shown in [Listing 4.9](#). Save this file in the root directory of your application under the name `data.json`.

```
{
  "books": [
    {
      "id": 1,
      "title": "JavaScript - The Comprehensive Guide",
      "author": "Philip Ackermann",
      "isbn": "978-3836286299",
      "rating": 5
    },
    {
      "id": 2,
      "title": "Clean Code",
      "author": "Robert Martin",
      "isbn": "978-0132350884",
      "rating": 4
    }
  ]
}
```

```

    },
    {
      "id": 3
      "title": "Design Patterns",
      "author": "Erich Gamma",
      "isbn": "978-0201633610",
      "rating": 5
    }
  ]
}

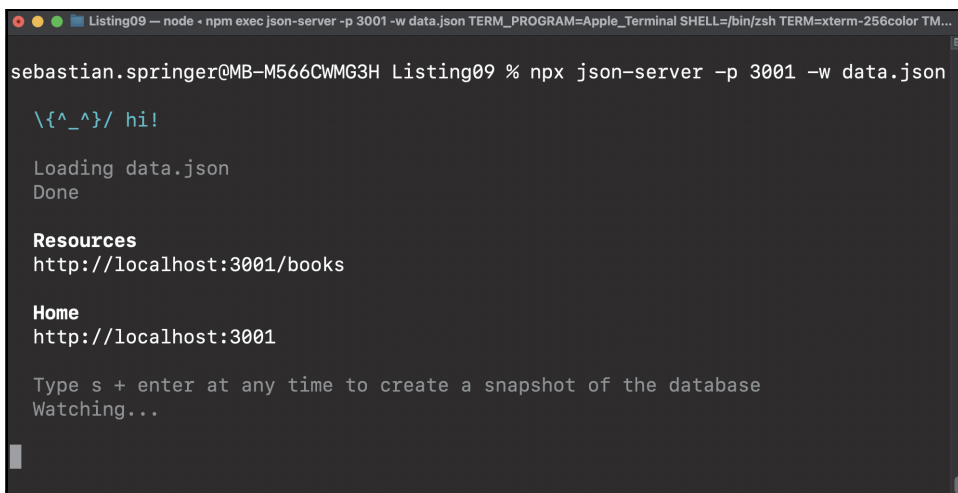
```

Listing 4.9 Server Data (data.json)

This file enables you to start the server using the `npx json-server -p 3001 -w data.json` command. You can then access the data via `http://localhost:3001/books`. You can specify the port with the `-p` option. The default port of the `json-server` is 3000; to avoid conflicts with the React dev server, you want to use port 3001 in this case. The `-w` option specifies that the file specified will be opened in watch mode and changes will take effect automatically.

In the console output, you should check if the `books` resource is listed as shown in [Figure 4.3](#). If it isn't, then there's a problem and the server does not deliver the data correctly. In this case, check the file and see if you have specified the correct path.

If you look into the file, you'll see an object structure in the top level. The keys determine the paths; in the example, that's the `books` key. Below this key you'll find an array of objects. You can access the objects all together, but also individually. When you open this address in the browser, you'll see a result like the one shown in [Figure 4.4](#).



```

sebastian.springer@MB-M566CWMG3H Listing09 % npx json-server -p 3001 -w data.json

\{^_^}/ hi!

Loading data.json
Done

Resources
http://localhost:3001/books

Home
http://localhost:3001

Type s + enter at any time to create a snapshot of the database
Watching...

```

Figure 4.3 Output of “json-server”

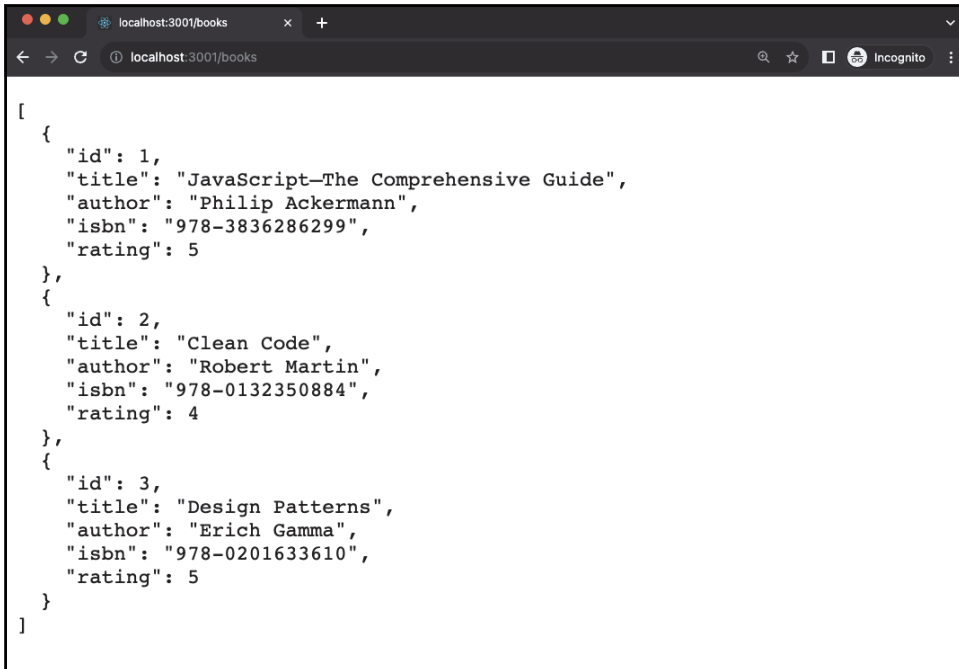


Figure 4.4 Accessing the Interface of “json-server” in the Browser

If you run the server on your system, you can customize your application to load and display the data from the server.

4.3.2 Server Communication via the Fetch API

Communicating with a web server is a classic side effect in a React application. For this reason, you place these requests not directly in the component function, but in an effect hook.

```
import { useState } from 'react';
import './BooksList.css';
import { useEffect } from 'react';

function BooksList() {
  const [books, setBooks] = useState([]);
  const [error, setError] = useState(null);

  useEffect(() => {
    fetch('http://localhost:3001/books')
      .then((response) => {
        if (!response.ok) {
          throw new Error('Request failed');
        }
      })
      .catch((error) => {
        setError(error);
      })
      .then((response) => {
        setBooks(response.json());
      });
  });
}
```

```

    }
    response.json();
  })
  .then((data) => {
    setBooks(data);
  })
  .catch((error) => setError(error));
}, []);

if (error !== null) {
  return <div>An error has occurred: {error.message}</div>;
} else if (books.length === 0) {
  return <div>No books found</div>;
} else {
  return (
    <table>
      <thead>
        <tr>
          <th>Title</th>
          <th>Author</th>
          <th>ISBN</th>
        </tr>
      </thead>
      <tbody>
        {books.map((book) => (
          <tr key={book.id}>
            <td>{book.title}</td>
            <td>{book.author}</td>
            <td>{book.isbn}</td>
          </tr>
        ))}
      </tbody>
    </table>
  );
}
}
}

export default BooksList;

```

Listing 4.10 Loading the Books Data from the Server (src/BooksList.jsx)

The `BooksList` component has the `books` state, where you store the data for display. You also define an additional error state, which serves as a repository for any error messages that may occur.

You call the `useEffect` function with a callback function and an empty dependency array so that the data is loaded only when the component is mounted. If you forget this second parameter, you won't notice it at first. But when you open the developer tools of your browser and look at the outgoing network requests, you'll notice that not just one request is sent, but hundreds of them in no time. That's because in this case you start a server request, and when it responds you update the state, which in turn triggers the Effect callback, and so on, so you're in an infinite loop. For this reason, you should look at the developer tools every now and then so that you'll notice any unusual behavior of your application during development.

Inside the Effect callback, you call the `fetch` function of the browser with the address of the server interface. The result is a `Promise` object that maps the asynchronous operation. You can use the `then` method of this object to access the response object of the request. There you can also check if everything was OK with the response by checking the `ok` property of the `response` object. If it contains the value `false`, you throw an exception in our case, which you'll handle later. If the response doesn't contain any error, you can process the response in the next step. You must keep in mind that HTTP is a streaming protocol, where the body of the response can be transmitted in several parts from the server to the browser.

For this reason, you must also consume the body separately. The browser's Fetch API provides you with the `json` method for this purpose, which is also asynchronous. It allows you to decode the response and interpret it as a JSON object. The resulting `Promise` object then contains the server's response, which you write to the component's state. If the request was successful, you'll see the book list in your browser.

Error Handling

What happens if the server doesn't respond or you mistyped the interface URL? If you don't provide for these cases, the browser will throw an exception and stops running your application. For this reason, the source code from [Listing 4.10](#) handles these types of errors at a general level. To the chain of calls of the `then` methods of the `Promise` objects, you add the call of the `catch` method. If an error occurs when requesting the server or decoding the response, the browser executes the callback function of the `catch` method and writes the `error` object to the `error` state of the component.

To display the error, you must check that the `error` state no longer has a value of `null` when you render the component, and then display the `message` property of the `error` object. The type and scope of information you display to your users in the event of an error is entirely up to you. You should generally use error information sparingly or be careful with the content of error messages so as not to reveal too much to potential attackers on your application.

You can test whether your error-handling routine works by terminating the server process or specifying an invalid path such as `booksX`. In both cases, your application does not render the list, but displays the corresponding error message.

Using “async”/“await”

When using promises, as in the example case with server requests, you often use the `async/await` feature of the browser. It allows you to use promises, but without callback functions. This means you can streamline your source code and make it easier to read. The key is to add the `async` keyword to the function in which you use promises. Then you can prefix each asynchronous operation with the `await` keyword and get the promise value directly. The browser takes care of waiting for the result and doesn't block the rest of your application either.

What you also need to know at this point is that an `async` function always returns a promise object. This fact makes integration with the Effect callback more difficult, as the React API dictates that this callback function should either return a function or nothing at all. You can still use `async/await`, but you have to create a helper construct for it. Listing 4.11 shows the customized source code of the `useEffect` call. The rest of the component remains unchanged.

```
useEffect(() => {
  (async () => {
    try {
      const response = await fetch('http://localhost:3001/books');
      if (!response.ok) {
        throw new Error('Request failed');
      }
      const data = await response.json();
      setBooks(data);
    } catch (error) {
      setError(error);
    }
  })();
}, []);
```

Listing 4.11 “useEffect” with “async”/“await” (src/BooksList.jsx)

To enable you to use `async/await` in the `useEffect` callback, you want to define an *immediately invoked function expression* (IIFE), an anonymous self-calling function in the form `(async () => {...})();`. This is a function that you define and call immediately. The `async` keyword allows you to use `await`. For error handling, you can then use a `try-catch` statement. The logic itself—calling `fetch`, checking and decoding the response, and setting the state and handling errors—remains the same. So if you switch to the browser with this customization, you will see no difference from the previous implementation. Which one you choose is up to you. But you should then consistently opt for your chosen solution instead of switching back and forth.

4.3.3 Things to Know about Server Communication

There are two more aspects that may be of interest to you in connection with server communication. First, there is the `proxy` feature, which allows you to redirect requests to a backend; and second, there is the use of environment variables, and in particular the specification of the server address. The information in the following sections refers to an application you've set up using Create React App. Other environments provide similar features, but you should look at the respective documentation for this.

Proxy

Currently you're communicating with the server interface at the address `http://localhost:3001/`. This causes problems in three places at once: In a productive application, unencrypted communication via the HTTP protocol is a no-go. The localhost host name also doesn't work in a production environment as it refers to the local system, and the port specification 3001 is also rather unusual and is replaced by the default port 80 in live operation. So you should avoid writing the address of the development system directly into the source code.

There are several approaches that help you solve this problem. One of the simplest variants is the use of the proxy feature. In this approach, you add the proxy configuration to the `package.json` file of your application. In your application, you just reference the server interface as a regular path in your application, and the `react-scripts` package takes care of redirecting requests to the interface. Not only does this solve the problem of the server address stored in the source code, but it also avoids potential difficulties with *cross-origin resource sharing* (CORS). Your browser disallows requests to remote systems if they are directed to a different origin—that is, a combination of scheme, host, and port. For the deployment of your application, however, this means that you also need a system that delivers the frontend and backend via the same origin.

To make the proxy work for the development environment, you need to add an appropriate entry to your `package.json` file. For the interface of the example, which is accessible at `http://localhost:3001/`, the entry looks as follows:

```
{
  ...
  "proxy": "http://localhost:3001"
}
```

Listing 4.12 Proxy Configuration in the `package.json` File

This customization allows you to specify only the server interface path in the components. The proxy receives these requests and forwards them to the server. [Listing 4.13](#) shows the customized call of the `useEffect` function in the `BooksList` component:

```

useEffect(() => {
  (async () => {
    try {
      const response = await fetch('/books');
      if (!response.ok) {
        throw new Error('Request failed');
      }
      const data = await response.json();
      setBooks(data);
    } catch (error) {
      setError(error);
    }
  })();
}, []);

```

Listing 4.13 Using the Proxy in the Application (src/BooksList.jsx)

But there's another and even more flexible solution to get rid of references to third-party systems in the code.

Environment Variables

Environment variables represent information that is provided on your system. The applications of the system can access this information and adjust their behavior. The advantage is that you can design the contents of the environment variables differently on each system on which you run your application, and you do not have to change the application's code to do so.

You can set environment variables in different ways. The simplest variant is to set it on the command line, where you also start the application. Another option is to use *.env* files.

Create React App already prepares the use of environment variables for you, so you can use them directly. By default, you can access the `NODE_ENV` variable. In it you specify whether you are currently in a development or production environment. You can also define any environment variables of your own. The only requirement is that the variable names start with `REACT_APP`. For example, for the case of the API server, the name of the variable could be `REACT_APP_API_SERVER`. You access the contents of the variables via `process.env`. [Listing 4.14](#) shows how to use the `REACT_APP_API_SERVER` variable in your component:

```

useEffect(() => {
  (async () => {
    try {
      const response = await fetch(

```



```
    `${process.env.REACT_APP_API_SERVER}/books`
  );
  if (!response.ok) {
    throw new Error('Request failed');
  }
  const data = await response.json();
  setBooks(data);
} catch (error) {
  setError(error);
}
})();
}, []);
```

Listing 4.14 Using Environment Variables in Components (src/BooksList.jsx)

In the case of the `BooksList` component, you prefix the path of the `fetch` call with the name of the environment variable in the `Effect` callback. In the next step, you take care of setting the environment variable. You can do this, for example, directly when starting the application with the following command:

```
REACT_APP_API_SERVER=http://localhost:3001 npm start
```

Listing 4.15 Starting the Application with Environment Variables

Note that setting environment variables works differently from system to system. For example, the command in [Listing 4.15](#) works on Unix systems. Much more comfortable and flexible is a combination of the `cross-env` package and an NPM script. For this purpose, you first need to install the package using the `npm install cross-env` command. Then you want to modify the `start` script in your `package.json` file as shown in [Listing 4.16](#):

```
{
  ...
  "scripts": {
    "start": "cross-env REACT_APP_API_SERVER=http://localhost:3001
      react-scripts start",
    ...
  }
}
```

Listing 4.16 Setting Environment Variables in the `package.json` File

This solution is convenient and works across different systems. However, this variant turns into a problem if you want to set several environment variables. In that case, the `package.json` file quickly becomes confusing, and here the use of a `.env` file is

recommended. You can see the contents of this file, which you place in the root directory of your application, in [Listing 4.17](#):

```
REACT_APP_API_SERVER=http://localhost:3001
```

Listing 4.17 Storing Environment Variables in a .env File

One aspect to keep in mind when using environment variables is that Create React App integrates the environment variables into the application at build time. So the environment variables are not analyzed regularly in a production application, but only once when you call `npm run build` to build the application.

In the next section, we'll introduce another alternative to the browser's Fetch API: the Axios library.

4.3.4 Server Communication with Axios

The Fetch API of the browser is the default tool when it comes to server communication. But especially for larger applications, this interface lacks some convenient features. Libraries such as Axios, which build on the Fetch API and add additional features, provide a workaround.

You install Axios in your application using the `npm install url axios` command. The additional installation of `url` package is required because all Webpack versions prior to version 5 included polyfills for node core packages. However, newer versions of Create React App use Webpack 5, which results in a bug when you use Axios. You can fix this by installing the URL package. Once installed, you can import Axios and use the library instead of the Fetch API in your components. [Listing 4.18](#) shows the deployment in the BooksList component:

```
import axios from 'axios';
...
useEffect(() => {
  (async () => {
    try {
      const { data } = await axios.get(
        `${process.env.REACT_APP_API_SERVER}/books`
      );
      setBooks(data);
    } catch (error) {
      setError(error);
    }
  })();
}, []);
```

Listing 4.18 Using Axios in a React Application (src/BooksList.jsx)

As you can see in the code, Axios handles the correct decoding of the JSON response as well as error handling at this point. Like the Fetch API, Axios uses promises, so you only need to make minor adjustments to your source code.

However, a few saved lines do not justify the use of an additional library like Axios. This library provides even more features, such as the configuration of instances. This has the advantage that you can centrally configure an instance and provide it with information such as the `baseUrl` or specific header fields and then use it anywhere in your application. [Listing 4.19](#) contains an example of such an Axios instance:

```
import { useState, useEffect } from 'react';
import './BooksList.css';
import axios from 'axios';

const client = axios.create({
  baseUrl: process.env.REACT_APP_API_SERVER,
});

function BooksList() {
  const [books, setBooks] = useState([]);
  const [error, setError] = useState(null);

  useEffect(() => {
    (async () => {
      try {
        const { data } = await client.get(`/books`);
        setBooks(data);
      } catch (error) {
        setError(error);
      }
    })();
  }, []);

  if (error !== null) {
    return <div>An error has occurred: {error.message}</div>;
  } else if (books.length === 0) {
    return <div>No books found</div>;
  } else {
    return (
      <table>...</table>
    );
  }
}
```

```
export default BooksList;
```

Listing 4.19 Using an Axios Instance (`src/BooksList.jsx`)

If you use the instance in more than one component, you can swap it out to a separate file and access it from multiple places in your application.

So far, you've combined the logic and display in your React components—but this quickly leads to the components becoming large and confusing. You can address this problem in different ways. *Container components* represent one option.

4.4 Container Components

React makes very few specifications about the structure and content of a component. However, if you follow the conventions, you should make sure that your components remain simple and clear and that a component is responsible for exactly one task only. React supports you here by keeping the creation of new components very lightweight. If you start from the `BooksList` component—that is, a component that displays a simple list—it takes care of loading the data, organizing the state, and displaying it.

So far, we have only used visual features to divide a component. Container components allow you to also differentiate between the logic and the presentation. The counterpart of a container component is a *presentational component*. Dan Abramov also distinguishes between *smart* and *dumb components* in one of his blog articles at <http://s-prs.co/v570506>. This designation alludes to the fact that presentational components are only supposed to take care of rendering; that is, they are supposed to be dumb in terms of the application logic. Container components on the other hand return a JSX structure, but this usually consists of only one or at least very few presentational components. The container component encapsulates the logic for the presentational component, carries the state in the form of its internal state, and performs the communication with the server. The information and functions that the presentational component needs for the display are passed to it as props by the container component.

The distinction between presentational and container components has the advantage that you clearly separate display and logic. This results in an improved reusability of the presentational components because the logic becomes interchangeable. Also, you can test the logic separately from the actual rendering.

For a better distinction, you can extend the file name of the container component with the name *container*.

In an application, you should normally also proceed according to this scheme and implement your components first. If you find that a component is getting too large or that you want to reuse certain parts of the application logic, you should separate the representation from the logic and insert containers.

4.4.1 Swapping Out Logic to a Container Component

For the implementation of container and presentational components, we start from the `BooksList` component:

```
import { useState, useEffect } from 'react';
import './BooksList.css';
import axios from 'axios';

const client = axios.create({
  baseURL: process.env.REACT_APP_API_SERVER,
});

function BooksList() {
  const [books, setBooks] = useState([]);
  const [error, setError] = useState(null);

  useEffect(() => {
    (async () => {
      try {
        const { data } = await client.get(`/books`);
        setBooks(data);
      } catch (error) {
        setError(error);
      }
    })();
  }, []);

  if (error !== null) {
    return <div>An error has occurred: {error.message}</div>;
  } else if (books.length === 0) {
    return <div>No books found</div>;
  } else {
    return (
      <table>
        <thead>
          <tr>
            <th>Title</th>
            <th>Author</th>
            <th>ISBN</th>
          </tr>
        </thead>
        <tbody>
          {books.map((book) => (
            <tr key={book.id}>
              <td>{book.title}</td>
              <td>{book.author}</td>
              <td>{book.isbn}</td>
            </tr>
          ))}
        </tbody>
      </table>
    );
  }
}
```

```

    ))}
  </tbody>
</table>
);
}
}

```

```
export default BooksList;
```

Listing 4.20 Initial Situation: The “BooksList” Component (src/BooksList.jsx)

When splitting the component, you first need to identify all the parts that have nothing to do with rendering and swap them out to a separate component. Choose *BooksList.container.jsx* as the file name. The component is called *BooksListContainer*. In [Listing 4.21](#), you can see the implementation of the container component:

```

import { useState, useEffect } from 'react';
import BooksList from './BooksList';
import axios from 'axios';

const client = axios.create({
  baseUrl: process.env.REACT_APP_API_SERVER,
});

function BooksListContainer() {
  const [books, setBooks] = useState([]);
  const [error, setError] = useState(null);

  useEffect(() => {
    (async () => {
      try {
        const { data } = await client.get(`/books`);
        setBooks(data);
      } catch (error) {
        setError(error);
      }
    })();
  }, []);

  return <BooksList error={error} books={books} />;
}

export default BooksListContainer;

```

Listing 4.21 Container Component (src/BooksList.container.jsx)

In this case, you cut out the state and effect hooks and pass on all the structures needed by the presentational component via props. For the `BooksList` component, these are the `error` and `books` states.

4.4.2 Integrating the Container Component

When you integrate a component, it doesn't matter to React what type of component it is—that is, whether it's a presentational or a container component. All components are specified to another component by a tag in the JSX. It also does not distinguish the interface to the outside system—that is, the props. Because you use `default exports` throughout for the components, when you include the component in `app`, you only need to modify the `import` statement in the `App` component, as shown in [Listing 4.22](#):

```
import './App.css';
import BooksList from './BooksList.container';

function App() {
  return (
    <div>
      <h1>Books management</h1>
      <BooksList />
    </div>
  );
}

export default App;
```

Listing 4.22 Integration of the Container Component (`src/App.jsx`)

Before you can test the new version of the application, you need to convert the presentational component.

4.4.3 Implementing the Presentational Component

You implement the presentational component as a function component that returns the JSX structure of the original `BooksList` component. To make sure that this will continue to work, you need to include the `books` array and the `error` object as props. The source code of the presentational component is shown in [Listing 4.23](#):

```
import PropTypes from 'prop-types';
import './BooksList.css';

function BooksList({ error, books }) {
  if (error !== null) {
    return <div>An error has occurred: {error.message}</div>;
  }
}
```

```

    } else if (books.length === 0) {
      return <div>No books found</div>;
    } else {
      return (
        <table>
          <thead>
            <tr>
              <th>Title</th>
              <th>Author</th>
              <th>ISBN</th>
            </tr>
          </thead>
          <tbody>
            {books.map((book) => (
              <tr key={book.id}>
                <td>{book.title}</td>
                <td>{book.author}</td>
                <td>{book.isbn}</td>
              </tr>
            ))}
          </tbody>
        </table>
      );
    }
  }
}

BooksList.propTypes = {
  books: PropTypes.arrayOf(
    PropTypes.shape({
      id: PropTypes.number.isRequired,
      title: PropTypes.string.isRequired,
      isbn: PropTypes.string.isRequired,
    })
  ),
  error: PropTypes.object,
};

```

```
export default BooksList;
```

Listing 4.23 Implementation of the Presentational Component (src/BooksList.jsx)

The only major customization here is the insertion of `PropTypes` for the presentational component, which you use to ensure that the correct data structures are passed to the component.

Tip

The division into containers and presentational components is a continuous process in an application. Do not try to divide your entire application into these categories at the very beginning of the development, but instead add containers where you need them.

4.5 Higher-Order Components

React components should generally be focused on exactly one purpose. Often, components then need to be extended by functionality that is not directly related to the display or is used in a similar way in several components. A classic example of this is the connection to a data source. One possibility is to use container components. Another, quite frequently used solution is to use so-called higher-order components (HOCs). This designation alludes to the higher-order functions from functional programming; this type of function is often found in JavaScript, for example, in array functions. These are functions to which functions are passed as arguments or that return functions themselves. The higher-order components in React follow a similar concept. A HOC consists of a function to which you pass a component and which in turn returns a component.

A HOC is a design pattern and not a concrete implementation of React. The pattern exploits the fact that components can be used like objects. HOCs are often used when the functionality of components is to be extended. They are used both within applications and in the creation of libraries. However, with the increasing adaptation of the Hooks API, HOCs are becoming less important as you can usually implement the functionality with a custom hook. You'll learn how this works in [Chapter 6](#).

4.5.1 A Simple Higher-Order Component

Before we integrate a HOC into a larger context, let's first look at a simple example: A `button` component is supposed to be extended with a `log` option. Basically, using a HOC is about adding functionality to the passed component. The `button` component has a `log` prop through which logging functionality is passed to it. The `button` component calls this functionality as a function. The HOC does nothing more than generate a new component consisting of the `button` component and a concrete `log` function. So you assign a fixed value to the `log` prop.

```
function Button({ log, onClick, title }) {  
  return (  
    <button  
      onClick={(e) => {  
        log(e);  
      }}  
    />  
  );  
}
```

```

        onClick(e);
      }}
    >
    {title}
  </button>
);
}

function withLogger(Component) {
  function log(item) {
    console.log('Logger: ', item);
  }

  return function (props) {
    return <Component log={log} {...props} />;
  };
}

const ButtonWithLogger = withLogger(Button);

function App() {
  return (
    <div>
      <ButtonWithLogger
        onClick={() => console.log('click handled')}
        title="Click me"
      />
    </div>
  );
}

export default App;

```

Listing 4.24 Simple Higher-Order Component

The component to be extended here is the `button` component. This component accepts a `log` function, a click-handler function, and the button label. The button component returns a button element whose label you set and in whose `onClick` prop you call the `log` function and the `onClick` function. The `log` function is passed to the component via the props. Such a logging facility is usually required in several places in an application. In addition, different implementations are used for different environments, such as the development or production environment. To avoid having to replace the implementation in all places, you can use a HOC and extend the components that need the logging feature accordingly. If the functionality is to be replaced later, you only need to adjust the HOC.

The `withLogger` function represents the HOC. This function receives a component as an argument and in turn returns a component, in this case in the form of a function component. This component renders the originally passed component and passes the `log` function to it. In the example, the `log` function writes the value passed to it to the browser console. However, an alternative implementation could also send the value to a logging server. A widely used convention states that a HOC should be preceded by the word `with` to identify the type of component at first glance.

When implementing HOCs, you should keep in mind that these functions should be *pure functions*—that is, functions without side effects that generate an output from a given input via reproducible rules. The input and output in this case are React components. Moreover, the HOC does not modify the passed component, but merely enriches it with additional functionality that is passed to the inner component in the form of props.

As you can see in the code example, the `withLogger` function is used like an ordinary JavaScript function, which means that in this case you can also control the behavior of the function with additional arguments.

The component you created through the `withLogger` function passes all props passed to it to the component it encapsulates. You can achieve this by having the wrapper component of the HOC itself accept props, and you pass them to the inner component through a destructuring statement.

One last point to note is that you shouldn't call the HOC within another component function. This would result in the component being recreated each time it's rendered. If you call the function outside the component function, the component is created once and merely redisplayed during each rendering process.

4.5.2 Integrating a Higher-Order Component in the BooksList Component

For the concrete implementation of a HOC in an application, we'll use the `BooksList` component. The logic for loading the data records gets swapped out to a HOC. This has the advantage that you can easily replace this HOC as a data provider later and, for example, use a WebSocket stream to load and update the data instead of an HTTP request to a backend. The name of the HOC is `withBooks`. It enriches the passed component with the `books` prop, which contains an array of data records, and the `error` prop, which contains either null or an error object. [Listing 4.25](#) contains the implementation of this method.

```
import { useEffect, useState } from 'react';
import axios from 'axios';

function withBooks(Component) {
  return function (props) {
    const [books, setBooks] = useState([]);
```

```

const [error, setError] = useState(null);

useEffect(() => {
  (async () => {
    try {
      const { data } = await axios.get(
        `${process.env.REACT_APP_API_SERVER}/books`
      );
      setBooks(data);
    } catch (error) {
      setError(error);
    }
  })();
}, []);

return <Component {...props} books={books} error={error} />;
};
}

export default withBooks;

```

Listing 4.25 HOC for Loading Data Records (src/withBooks.jsx)

You implement the HOC as a function named `withBooks`. The function in turn returns an anonymous function that represents the component that is responsible for loading the data and managing the state. For this purpose, you create a `books` and an `error` state and load the data from the server via an effect hook. The effect hook takes care of setting the state and error handling. The return value of the anonymous component is the passed component, to which you pass all passed props. In addition, you insert the `books` and `error` props.

4.5.3 Integrating the Higher-Order Component

You can take the inner component—the `BooksList` component—directly from [Listing 4.23](#). Because the interface via the props already corresponds to what the HOC requires, no adjustments are necessary.

When integrating the HOC in the application, the rule is that you shouldn't call the HOC function within the function component; otherwise, it would be created again for each rendering process.

```

import './App.css';
import BooksList from './BooksList';
import withBooks from './withBooks';

```

```
const BooksListWithBooks = withBooks(BooksList);
```

```
function App() {  
  return (  
    <div>  
      <h1>Books management</h1>  
      <BooksListWithBooks />  
    </div>  
  );  
}
```

```
export default App;
```

Listing 4.26 Integration of the higher-order component (src/App.jsx)

As you can see in [Listing 4.26](#), you run the `withBooks` function outside the actual component. The `App` component then renders the `BooksListWithBooks` component within its JSX structure. When you switch to the browser, you shouldn't notice any visible change in the display, as it doesn't matter whether you choose a container component or a HOC. The only issue here is how and where you swap out the logic of your application. In addition to these two approaches, using render props is another method of removing logic from a component and reusing it elsewhere in the application.

4.6 Render Props

Like HOCs, *render props* denote an architectural pattern in React and not a feature of the library. Here, you create a component that accepts a prop named `render` and then executes it in its own function body. You can then pass additional information during this execution. We'll demonstrate the implementation of render props with a small example first, before we integrate it into the sample application. For demonstration purposes, we'll first use the logger example again. [Listing 4.27](#) contains the source code of the example:

```
function Button({ log, onClick, title }) {  
  return (  
    <button  
      onClick={e => {  
        log(e);  
        onClick(e);  
      }}  
      >  
        {title}  
      </button>  
    );  
}
```

```

function log(item) {
  console.log('Logger: ', item);
}

function Logger({ render }) {
  return render(log);
}

function App() {
  return (
    <div>
      <Logger
        render={(log) => (
          <Button
            log={log}
            onClick={() => console.log('click handled')}
            title="Click me"
          />
        )}
      />
    </div>
  );
}

export default App;

```

Listing 4.27 Implementation of Render Props

In [Listing 4.27](#), the parts of the example that differ from the HOC example are marked in bold. The `logger` component replaces the `withLogger` HOC in this case. The `Logger` component assumes that the `render` prop passed returns a component that the function component itself can return. The `log` function is passed as an argument to the function from the `render` prop. The purpose behind this implementation only becomes clear when you imagine that the `Logger` component and the `log` function are stored in a separate file and are used in different places in the application.

In the `App` component itself, you render the `Logger` component and then specify a function in the `render` prop that receives the `log` function and renders the `Button` component.

4.6.1 Alternative Names for Render Props

Render props get their name from the naming of the prop that determines the rendering. However, this name is not fixed; you can choose it freely. The name `render` is merely a relatively widespread convention here. Similarly, the name `children` is also used frequently. However, this prop name has a special feature: it refers to the *children*

of a component—that is, the elements located between the opening and closing tags. So you can use this prop in two ways. In the simpler case, you just rename the `render prop` to `children` and don't have to change anything else. [Listing 4.28](#) shows how you can use the `children` prop to access a function that you have defined as a child element of the `logger` component:

```
function Button({ log, onClick, title }) {
  return (
    <button
      onClick={(e) => {
        log(e);
        onClick(e);
      }}
      >
      {title}
    </button>
  );
}

function log(item) {
  console.log('Logger: ', item);
}

function Logger({ children }) {
  return children(log);
}

function App() {
  return (
    <div>
      <Logger>
        {(log) => (
          <Button
            log={log}
            onClick={() => console.log('click handled')}
            title="Click me"
          />
        )}
      </Logger>
    </div>
  );
}

export default App;
```

Listing 4.28 Render Props with “children” Elements

The only adjustment to the `logger` component consists of renaming the prop from `render` to `children`. During the integration, you implement an arrow function between the opening and closing `logger` tags, which you can access through the `children` prop of the `logger` component. This arrow function is passed the `log` function, which you can then in turn pass to the `button` component.

Compared to the other implementations, this is probably one of the most elegant solutions. However, the choice of the specific implementation ultimately remains a matter of taste.

4.6.2 Integrating the Render Props into the Application

For the integration of the render props, we use the last described variant with the `children` prop and the implementation of the `render` callback function within the component. The `withBooks` component is replaced by a new component called `BooksLoader`. The source code of the component remains almost unchanged. Only the props and the returned JSX structure need to be adjusted. [Listing 4.29](#) contains the source code:

```
import { useEffect, useState } from 'react';
import axios from 'axios';

function BooksLoader({ children }) {
  const [books, setBooks] = useState([]);
  const [error, setError] = useState(null);

  useEffect(() => {
    (async () => {
      try {
        const { data } = await axios.get(
          `${process.env.REACT_APP_API_SERVER}/books`
        );
        setBooks(data);
      } catch (error) {
        setError(error);
      }
    })();
  }, []);

  return children(books, error);
}
```

```
export default BooksLoader;
```

Listing 4.29 The “BooksLoader” Component with Render Props (src/BooksLoader.jsx)

In the `BooksLoader` component, you manage the state, load the data from the server, and pass it to the function you pass to the component via the `children` prop. In the `App` component, you now include the `BooksLoader` component instead of the HOC. You can see what the corresponding source code looks like in [Listing 4.30](#):

```
import './App.css';
import BooksList from './BooksList';
import BooksLoader from './BooksLoader';

function App() {
  return (
    <div>
      <h1>Books management</h1>
      <BooksLoader>
        {(books, error) => <BooksList books={books} error={error} />}
      </BooksLoader>
    </div>
  );
}

export default App;
```

Listing 4.30 Integration of the Component with Render Props

Between the opening and closing `BooksLoader` tags, you define the function called by the `BooksLoader` component. The function returns the `BooksList` component, which in turn receives the data to display and an optional error object.

With these adjustments, you've now become familiar with both HOCs and render props and seen that you can easily interchange the two.

4.7 Context

Quick Start

The Context API allows you to access centrally stored information independently of the component hierarchy and without passing the information to child components via props.

You can create the context using the `createContext` function. The resulting object gives you a provider component in whose `value` prop you can store any structures, from simple values to objects and arrays to functions.

You can access these values from child components of the provider using the `useContext` function in conjunction with the context object.

The Context API of React helps you provide structures in the component tree without the need to use props to pass them through each layer from source to target. The use of the Context API is particularly suitable for larger applications in which different places access information. The interface is used by external packages (such as the React router or Redux) as well as in application development. Thus the Context API enables you to decouple your components more and avoid passing around props. While this means that component function signatures are simplified as you save props, it also means that complexity within your application may increase due to the use of different contexts.

In general, you should only use the Context API if you use a particular piece of information in multiple places within your application or if the source of a piece of information is several layers away from its actual use.

4.7.1 The Context API

The Context API of React provides a set of functions and components. The `createContext` function plays a central role as you use it to create a context. You can pass a default value to this function when you call it, which will be used by React if you don't assign an explicit value in your application.

The object returned by the `createContext` method contains two properties—`Provider` and `Consumer`. Both properties can be used as components. The `Provider` property can be used to set a value for the context. The `Consumer` property is used for read access to the context, where you must use the provider component to work with the context. However, you no longer use the `Consumer` component in a modern React application; instead, you access the context via the `useContext` function of the Hooks API.

Normally you generate the context object in a separate file using the `createContext` function. You can pass the context object a default value that React will use if no value is assigned elsewhere. The context can contain any number of objects, which are then available to you in the scope of the context. You use the `Provider` component to place the context in the component tree. Then you can access the context from all child components of the provider. Typically, you place the context near the root component of your application.

To illustrate the use of the Context API, let's first look at a simple example before we add the context to a larger application.

Creating a “Context” Object

First, you create a new file in your application called *Context.js*. The contents of this file are shown in [Listing 4.31](#):

```
import { createContext } from 'react';

const Context = createContext(0);
export default Context;
```

Listing 4.31 Creating a “Context” Object (src/Context.js)

The context is initialized with the value 0 in this case. You can access the resulting object in your application. This happens, for example, in the App component.

Integrating the Context

The Context object provides you with two components, `Provider` and `Consumer`. The `Provider` component provides the context for the component tree. In [Listing 4.32](#), you can see the integration into the App component:

```
import { useState } from 'react';
import Context from './Context';

function App() {
  const [counter, setCounter] = useState(0);

  function increment() {
    setCounter((prevState) => prevState + 1);
  }

  return (
    <Context.Provider value={counter}>
      <button onClick={increment}>increment</button>
    </Context.Provider>
  );
}

export default App;
```

Listing 4.32 Integrating the Context in the Application (src/App.jsx)

In this example, you create a counter state in the App component. The `increment` function allows you to increase the value of the counter by the value 1. In the JSX structure of the component, you use the `Provider` component of the Context object and assign the counter value as a value using the `value` prop. In the click-handler function of the button element, you call the `increment` function so that a click on the button increases the counter by one. Currently, this action doesn’t yet affect the display. In the next step, you implement another component that accesses the context and displays the value.

Read Access to the Context

For read access to the context, you use the `useContext` function. To display the counter value, create a new component called `Counter` and save it in a file called `Counter.jsx`:

```
import { useContext } from 'react';
import Context from './Context';

function Counter() {
  const value = useContext(Context);

  return <div>Counter: {value}</div>;
}

export default Counter;
```

Listing 4.33 Read Access to the Context with the “Context.Consumer” Component

As you can see in [Listing 4.33](#), you use the `useContext` function in combination with the context object you created via the `createContext` function. This gives the hook function access to the provider closest in the component tree and makes the value available to you. Now you still need include the component in the `App` component so that it gets displayed:

```
import { useState } from 'react';
import Context from './Context';
import Counter from './Counter';

function App() {
  const [counter, setCounter] = useState(0);

  function increment() {
    setCounter((prevState) => prevState + 1);
  }

  return (
    <Context.Provider value={counter}>
      <Counter />
      <button onClick={increment}>increment</button>
    </Context.Provider>
  );
}

export default App;
```

Listing 4.34 Integrating the “Counter” Component (src/App.jsx)

When you switch to the browser with this configuration, you can click on the button and the counter component will display the updated value. The most important difference is that you can access this value without passing it as a prop to the component.

4.7.2 Using the Context API in the Sample Application

This introductory example represented a simple use of the Context API. However, you can implement much more complex solutions with the context. So you can not only store simple values (like numbers in this example) in the context, but also objects, arrays, or functions. Next, you create a context where you store the array structure that the `useState` function returns to you. This allows you to access the state at various points, but also to modify it. This context is responsible for the management of book data records. In [Listing 4.35](#), you can see the source code that takes care of creating the context:

```
import { createContext } from 'react';

const BooksContext = createContext([null, () => {}]);

export default BooksContext;
```

Listing 4.35 Creating the Books Context (src/BooksContext.jsx)

By default, you set the context to an array with the `null` elements and an empty arrow function. These represent the two elements from `useState`. In the next step, you implement a component that creates the state, connects it to the context, and provides the context's provider. You name this file *BooksProvider.jsx*. The corresponding source code can be found in [Listing 4.36](#):

```
import { useState } from 'react';
import BooksContext from './BooksContext';

function BooksProvider({ children }) {
  const [books, setBooks] = useState([]);

  return (
    <BooksContext.Provider value={[books, setBooks]}>
      {children}
    </BooksContext.Provider>
  );
}

export default BooksProvider;
```

Listing 4.36 Creating “BooksProvider” (src/BooksProvider.jsx)

Here you can see another use of the `children` prop of a component. This implementation allows you to integrate the `BooksProvider` component into your component tree, and React will take care of ensuring that the elements you insert between the opening and closing tags are correctly placed as child elements in the component and thus have access to the context.

Now you need to include the `BooksProvider` component in your application. As mentioned earlier, you usually place the context near the root component, or in this case, just inside the `App` component, as shown in [Listing 4.37](#):

```
import BooksProvider from './BooksProvider';
import BooksList from './BooksList';

function App() {
  return (
    <BooksProvider>
      <BooksList />
    </BooksProvider>
  );
}

export default App;
```

Listing 4.37 Integration of “BooksProvider” into the Application (src/App.jsx)

The `BooksList` component is the first component in the tree that can access the context. As before, your task is to display the frame of the list and iterate over the individual data records. [Listing 4.38](#) shows the source code of the component:

```
import { useEffect, useContext } from 'react';
import BooksContext from './BooksContext';
import BooksListItem from './BooksListItem';
import './BooksList.css';
import axios from 'axios';

function BooksList() {
  const [books, setBooks] = useContext(BooksContext);
  useEffect(() => {
    (async () => {
      const { data } = await axios.get(
        `${process.env.REACT_APP_API_SERVER}/books`
      );
      setBooks(data);
    })();
  }, []);
}
```

```
return (  
  <table>  
    <thead>  
      <tr>  
        <th>Title</th>  
        <th>Author</th>  
        <th>ISBN</th>  
        <th></th>  
      </tr>  
    </thead>  
    <tbody>  
      {books.map((book) => (  
        <BooksListItem key={book.id} book={book} />  
      ))}  
    </tbody>  
  </table>  
)  
);  
}
```

```
export default BooksList;
```

Listing 4.38 “BooksList” Component with Access to the Context (src/BooksList.jsx)

The `BooksList` component uses the `useContext` function to gain access to the context. It loads the data from the server within an effect hook and writes it to the context using the `setBooks` function. In the JSX structure, the component iterates over the `books` array and creates new instances of the `BooksListItem` component for each data record. Here you pass the respective data record to the component for display. The code of the `BooksListItem` component is shown in [Listing 4.39](#):

```
import { useContext } from 'react';  
import BooksContext from './BooksContext';  
import produce from 'immer';  
import { StarBorder, Star } from '@mui/icons-material';  
  
function BooksListItem({ book }) {  
  const [, setBooks] = useContext(BooksContext);  
  
  function handleRate(id, rating) {  
    setBooks((prevState) => {  
      return produce(prevState, (draftState) => {  
        draftState.map((book) => {  
          if (book.id === id) {  
            book.rating = rating;  
          }  
        })  
      })  
    })  
  }  
}
```

```

        return book;
      });
    });
  });
}

return (
  <tr>
    <td>{book.title}</td>
    <td>{book.author ? book.author : 'Unknown'}</td>
    <td>{book.isbn}</td>
    <td>
      {new Array(5).fill('').map((item, i) => (
        <button
          className="ratingButton"
          key={i}
          onClick={() => handleRate(book.id, i + 1)}
        >
          {book.rating < i + 1 ? <StarBorder /> : <Star />}
        </button>
      ))}
    </td>
  </tr>
);
}

export default BooksListItem;

```

Listing 4.39 “BooksListItem” Component for Displaying the Individual Data Records (src/BooksListItem.jsx)

The component receives all information for display from its parent component. This means that no access to the context is required. However, the component includes the possibility to rate the books as well. This means that the component must also manipulate the data records. Previously, this task was performed by the parent component, which was also responsible for the state. In this case, you can get the `setBooks` function from the context, and the `BooksListItem` component can change the data itself. With the resulting state change, React updates the display, and the changed rating becomes visible to users.

When loading the `setBooks` function from the context, you see a special feature of the destructuring statement for arrays: if you aren’t interested in the first array element, you can put a comma at the beginning and access the second element directly. This omission of elements works for any number, but quickly becomes confusing, so you shouldn’t use this feature too excessively.

When creating the rating buttons, you'll see another feature, but it isn't React-specific: you use the array constructor to create a new array with five memory locations. In the next step, these will be filled with empty strings so that you can insert `button` elements via the `map` method in the final step. These elements call the `handleRate` function in their click handler, which takes care of communicating with the context.

After examining the somewhat more extensive Context API, we now come to a practical and manageable feature of React: fragments.

4.8 Fragments

When designing a React component, there are rules you must follow. One of the most important rules is that a component returns either exactly one root element or an array of elements. Thus, it isn't possible to render multiple parallel elements on the root layer of a component.

Often in such a case, you insert an additional `div` element to follow the rule. However, additional elements also mean that the DOM—the tree of HTML elements that the browser works with—will grow. This isn't yet a problem with individual elements, but with extensive component trees such additional elements become very important. React provides the *fragment component* for such cases. This component, introduced with version 16.2, is a container element that React does not render, but uses only for internal structuring.

For fragments, you can use two different notations. In the long notation, you use the `React.Fragment` component. The short notation consists only of an empty tag. In [Listing 4.40](#), you can see that the `h1` element with the heading and the `BooksProvider` are located at the top level of the component hierarchy. Instead of a `div` element, you can use a fragment here to return only a root element.

```
import BooksProvider from './BooksProvider';
import BooksList from './BooksList';

function App() {
  return (
    <>
      <h1>Books management</h1>
      <BooksProvider>
        <BooksList />
      </BooksProvider>
    </>
  );
}

export default App;
```

Listing 4.40 Using Fragments in the Application (`src/App.jsx`)

When you switch to the developer tools in the browser with this customization, you will see that the fragment is not rendered, nor is it translated to another HTML element.

4.9 Summary

In this chapter, you mainly dealt with the reusability of components and the swapping out of logic from components:

- You learned about the lifecycle of a component and how to influence its different sections using the effect hook.
- You learned how to communicate with a web server to use information in your application.
- *Container components* enable you to outsource logic from the components of your application.
- *Higher-order components* are a means of adding additional functionality to your components. These functions accept components as inputs and return extended components.
- Another means of increasing reusability within an application is via *render props*, where you pass a component to another component, extend it, and finally render it.
- The *Context API* allows access to global values from multiple places in the application without passing the information through each level of the component tree.
- *Fragments* are a means of grouping elements without having to insert an explicit container element. Fragments are not rendered.

In the next chapter, you'll learn more about the class components of React. This is the second method to implement components in React besides the function components we have used so far.

Chapter 5

Class Components

In a modern React application, function components are used by default. Although this design feature has nearly replaced the previously very widely used class components, class components still exist and will continue to be supported by React for quite some time, according to its developers. So if you don't need to work with class components because of an existing application, and you don't plan to introduce the deprecated class components in your application, you can skip this chapter and go directly to [Chapter 6](#), where you'll learn more about the Hooks API that was the cause of the demise of class components.

5.1 Class Components in React

Quick Start

A *class component* consists of a JavaScript class derived from `React.Component` and has at least one `render` method. In addition to this, there is a set of methods you can use to handle the lifecycle of the component. You can use the `state` property of the class to get read access to the component's state. With `setState`, you write the state. [Listing 5.1](#) contains an example of a class component:

```
import React from 'react';

class BooksList extends React.Component {
  state = { books: [] };

  async componentDidMount() {
    const response = await fetch(
      `${process.env.REACT_APP_API_SERVER}/books`
    );
    const data = await response.json();
    this.setState({ books: data });
  }

  render() {
    return (
      <ul>
```

```
    {this.state.books.map((book) => (  
      <li key={book.id}>{book.title}</li>  
    ))}  
  </ul>  
);  
}  
}  
  
export default BooksList;
```

Listing 5.1 Example of a Class Component

Class components get their name from the fact that they are implemented as JavaScript classes. In contrast to function components, they are more clearly structured. Especially if you are switching to React from other frameworks, you'll find your way around here much faster with class components than with function components. Class components were the original type of components in React and were the only way a component could manage its own state and lifecycle and access context prior to the introduction of the Hooks API.

5.2 Basic Structure of a Class Component

For a class component, you define a new JavaScript class that you derive from `React.Component`. This base class makes sure that the class becomes a component. The same rules and recommendations apply for handling class components as for function components. So the name of your class component must start with a capital letter, and the `render` method of the class must return a JSX structure with exactly one root element or an array of React elements. You should also make sure that you implement only one component per file and that the name of the file matches that of the component.

In [Listing 5.2](#), you can see a minimal version of a class component consisting of only one class with a `render` method:

```
import React from 'react';  
  
class App extends React.Component {  
  render() {  
    return (  
      <div>  
        <h1>Books management</h1>  
      </div>  
    );  
  }  
}
```

```
    }  
  }  
}
```

```
export default App;
```

Listing 5.2 Minimal Version of a Class Component (src/App.jsx)

It's always been possible to implement this type of component as a function component. This was even recommended because function components have significantly less overhead compared to class components. Class components only become relevant if they have their own state or lifecycle methods, which you can now also reproduce in function components via the `useState` and `useEffect` functions.

But before we delve into those topics, let's first look at how props should be handled in a class component.

5.3 Props in a Class Component

In a class component, you can use the `props` property to access the props passed from the parent component. And once again, the same principle applies: props can accept any data type, from simple numbers or strings to objects, arrays, and functions. [Listing 5.3](#) shows how you can access the props in a class component:

```
import React from 'react';  
  
class BooksListItem extends React.Component {  
  render() {  
    const { book } = this.props;  
    return (  
      <tr>  
        <td>{book.title}</td>  
        <td>{book.author}</td>  
        <td>{book.isbn}</td>  
      </tr>  
    );  
  }  
}
```

```
export default BooksListItem;
```

Listing 5.3 Accessing Props in a Class Component (src/BooksListItem.jsx)

If you include the `BooksListItem` component in your application using the `<BooksListItem book={book} />` tag, React allows you to access the `book` prop via `this.props.book` within the class component. By using this prop in more than one place

in the `render` method, you can save yourself some typing work by extracting the information into a variable via a destructuring statement.

5.3.1 Defining Prop Structures Using PropTypes

If you use JavaScript, you have the option to use PropTypes to define the structure of your props and thus make sure that it is adhered to. This is similar to using function components. You define PropTypes as a static property named `propTypes` for the class component:

```
import React from 'react';
import PropTypes from 'prop-types';

class BooksListItem extends React.Component {
  static propTypes = {
    book: PropTypes.shape({
      id: PropTypes.number.isRequired,
      title: PropTypes.string.isRequired,
      isbn: PropTypes.string.isRequired,
      rating: PropTypes.number.isRequired,
    }).isRequired,
  };

  render() {
    const { book } = this.props;
    return (
      <tr>
        <td>{book.title}</td>
        <td>{book.author}</td>
        <td>{book.isbn}</td>
      </tr>
    );
  }
}

export default BooksListItem;
```

Listing 5.4 Definition of PropTypes as Static Property of the Class (`src/BooksListItem.jsx`)

Listing 5.4 shows how to define PropTypes as a static property directly in the class. Alternatively, you can define them outside the class structure, as shown in Listing 5.5:

```
import React from 'react';
import PropTypes from 'prop-types';
```

```

class BooksListItem extends React.Component {
  render() {...}
}

BooksListItem.propTypes = {
  book: PropTypes.shape({}).isRequired,
};

export default BooksListItem;

```

Listing 5.5 Definition of PropTypes Outside the Class Component (src/BooksListItem.jsx)

Both variants are equivalent. However, the second variant is somewhat neater as especially with more extensive PropTypes the readability of the class will suffer. Regardless of which of the two variants you choose, however, if you violate the rules you have defined in PropTypes, then you'll receive a warning in the browser console.

5.3.2 Default Values for Props

In some cases, you specify that your component can be affected via props, but these props do not have to be specified. In this case, you can set default values for such optional props to create a defined starting point. You set these default props using the static `defaultProps` property. [Listing 5.6](#) shows the implementation of a headline component that accepts an optional `title` prop. If you don't pass it when including it, React uses the `Default heading` default value and displays it:

```

import React from 'react';
import PropTypes from 'prop-types';

class Headline extends React.Component {
  static defaultProps = {
    title: 'Default heading',
  };

  render() {
    return <h1>{this.props.title}</h1>;
  }
}

Headline.propTypes = {
  title: PropTypes.string,
};

export default Headline;

```

Listing 5.6 Default Values for Props (src/Headline.jsx)

In addition to the `render` method and the props, the basic functionality of a class component also includes the state of the component, which you can use to manage its internal state.

5.4 State: The State of the Class Component

The state of a component contains data that's needed for the display or to control the display. The component has control over the state and can modify it in any way. As with the function components, access to the state is divided into two parts: you can use the `state` property of the class to access the state in read-only mode, and the `setState` method to manipulate the state. Each time you make a change, React renders the component again.

You can set the initial value of the state in two ways, depending on whether this operation is static or dynamic.

5.4.1 Initializing the State via the State Property of the Class

A component is basically an ordinary JavaScript class, and as such it allows the use of properties. By deriving the component class from `React`, some methods (such as `render`) and also some properties (like the `state` property) receive a special meaning. [Listing 5.7](#) shows how to use the `state` property to initialize the component's state. You'll also see how to use the state of the class component for display.

```
import React from 'react';
import BookListItem from './BookListItem';

const initialBooks = [
  {
    id: 1,
    title: 'JavaScript - The Comprehensive Guide',
    author: 'Philip Ackermann',
    isbn: '978-3836286299',
    rating: 5,
  }
];

class BookList extends React.Component {
  state = {
    books: initialBooks,
  };

  render() {
```

```

    return (
      <table>
        <thead>
          <tr>
            <th>Title</th>
            <th>Author</th>
            <th>ISBN</th>
          </tr>
        </thead>
        <tbody>
          {this.state.books.map((book) => (
            <BooksListItem book={book} key={book.id} />
          ))}
        </tbody>
      </table>
    );
  }
}

```

```
export default BooksList;
```

Listing 5.7 Initialization of the State and Read Access (src/BooksList.jsx)

You can define the initial state directly via the `state` property of the class and assign an appropriate value to it. The disadvantage of this approach is that you have no dynamics here. So if you want to calculate the initial state, you have to switch to the constructor of the class, as you'll learn in the next section.

Inside the `render` method, you can see how to access the state in read-only mode. Again, follow the JavaScript rules and access the state property using `this.state`.

5.4.2 Initializing the State in the Constructor

The first method executed in a class component is the constructor, which allows you to perform initialization tasks for your component. But this isn't the right place to communicate with a server to dynamically load data; you'll learn how to do this when we describe the lifecycle of a class component. In the constructor, however, you can initialize the state of your component and use methods of the component. Also, you can use the props that were passed to your component. To demonstrate how the constructor works, we'll use the `Headline` component for this example. [Listing 5.8](#) shows the implementation of this component:

```

import React from 'react';
import PropTypes from 'prop-types';

```

```
class Headline extends React.Component {
  static defaultProps = {
    title: 'Default heading',
  };

  state = {
    title: '',
  };

  constructor(props) {
    super(props);

    this.state.title = props.title;
  }

  render() {
    return <h1>{this.state.title}</h1>;
  }
}

Headline.propTypes = {
  title: PropTypes.string,
};

export default Headline;
```

Listing 5.8 Creating the State Dynamically in the Constructor (src/Headline.jsx)

Note that you must run the parent constructor with `super(props)` when using the constructor of a class component. If you omit this line, you'll get a warning during the compile process and then a `ReferenceError` exception in the browser. In JavaScript, you must call the parent constructor before you can use `this`.

By default, React passes the props as an object to the constructor during the creation of a new component instance. You can also access the state via `this.state`. Thus, it's possible to dynamically fill the state once when the component instance is created. In that case, you want to use the `title` prop and assign it to the `title` property of the state. If you don't set the prop when you include the component, React uses the default value you set previously.

You're still missing one important building block for the implementation of class components: access to the individual stages of the lifecycle.

5.5 The Component Lifecycle

When you implement components, situations can occur in which you have to intervene in the component's lifecycle at different points in time. Typical examples include the creation of the component in the constructor, the update of props or states, or the unmounting of the component. Although the stations in the lifecycle of a class component are the same as in a function component, with class components, you have more intervention options. shows a graphical representation of the different lifecycle hooks available to you.

The lifecycle of a class component is divided into different sections:

- In the first section—the *mounting* section—the component is created and mounted in the application so that it becomes visible.
- The *updating* section describes the time during which the component is visible and users can interact with it. The main events that occur here are the changes from outside the component to the props and inside a component to the state.
- The last section is the *unmounting* section, which is triggered when the component is removed. This section is mainly about tidying up.

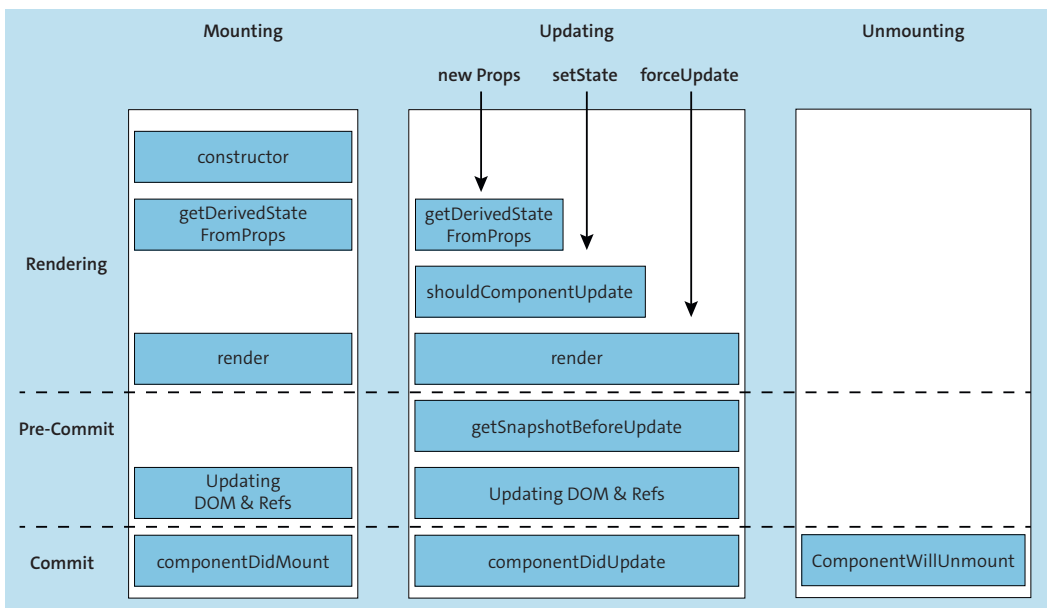


Figure 5.1 Lifecycle of a Class Component in React

Different hooks are available in each section. These are methods that you implement and that are executed as soon as the component is in the respective section. Each section is in turn divided into up to three different phases. The phases have different

characteristics when it comes to accessing the component and creating side effects as well as accessing the DOM:

- The *render* phase is responsible for rendering the component. The execution of the `render` method marks the end of this phase. This phase has no relevance for the unmounting section. The methods of this phase must not have any side effects because they can be stopped, aborted, and restarted by React.
- In the *precommit* phase, you can access the DOM of the component in read-only mode. This phase exists only in the updating section of the component lifecycle.
- The final phase of a component's lifecycle is the *commit* phase. It exists in every section and allows you to access the DOM. In addition, side effects, such as communication with a server, can be performed and updates can be scheduled.

To demonstrate the different lifecycle hooks, you can implement a `timer` component that counts up time using the different lifecycle hooks. For this example, assume that you have a newly created React application that consists of two components, `App` and `Timer`:

```
import React from 'react';
import './App.css';
import Timer from './Timer';

class App extends React.Component {
  state = {
    time: 0,
  };

  handleClick() {
    this.setState({ time: Math.floor(Math.random() * 10) });
  }

  render() {
    return (
      <div className="App">
        <Timer time={this.state.time} />
        <button onClick={() => this.handleClick()}>set</button>
      </div>
    );
  }
}

export default App;
```

Listing 5.9 Initial Component (`src/App.jsx`)

As you can see in [Listing 5.9](#), the `App` component has its own state, which contains the number `time`. This is passed to the `timer` component in the `render` method via the `time` prop. There's also a button that can reset the value through its click handler. Here, keep in mind that you lose the context inside an event-handler function. So if you were to pass the `handleClick` method directly to the `onClick` prop, the `this` inside the method would no longer point to the component instance. For this reason, you wrap the function call in an arrow function, which ensures that you keep the context.

5.5.1 Constructor

You use the constructor to perform the initialization of the component. As you already know, the constructor receives the passed props as an argument. Before you can use `this` in the constructor, you must call the parent constructor via the `props` object. Within the constructor, you can access the state via the `state` property and set it directly at this point without using the `setState` method:

```
import React from 'react';

export default class Timer extends React.Component {
  constructor(props) {
    console.log('Constructor');
    super(props);
    this.state = {
      time: 0,
    };
  }
}
```

Listing 5.10 Constructor of the “timer” Component (`src/Timer.jsx`)

In [Listing 5.10](#), you first write to the console the information that the constructor has been called. Finally, you set the `time` property of the `state` object to its initial value of 0.

5.5.2 “getDerivedStateFromProps”

The static `getDerivedStateFromProps` method is called on each change. This method replaces the `componentWillReceiveProps` method and is rarely used. The method is used to determine if the props have changed and if the state needs to be adjusted. In the `getDerivedStateFromProps` method, you have access to the `props` and the `state` object. The method returns a new state object or `null` if the state remains unchanged.

```
import React from 'react';

export default class Timer extends React.Component {
```

```
constructor(props) {  
  console.log('Constructor');  
  super(props);  
  this.state = {  
    initial: 0,  
    time: 0,  
  };  
}  
  
static getDerivedStateFromProps(props, state) {  
  console.log('getDerivedStateFromProps');  
  if (props.time !== state.initial) {  
    return {  
      initial: props.time,  
      time: props.time,  
    };  
  }  
  return null;  
}  
}
```

Listing 5.11 Implementation of the “getDerivedStateFromProps” Method (src/Timer.jsx)

In the example in [Listing 5.11](#), you add the `initial` property to the state. You use this property to check if the props have changed. In the `getDerivedStateFromProps` method, you first output the information that the function was called. If the props have changed, you set the time and initial values to the passed value.

5.5.3 “render”

If you don’t implement the `render` method, React cannot display the component. This component must return a JSX structure, which is then rendered by React.

```
import React from 'react';  
  
export default class Timer extends React.Component {  
  constructor(props) {...}  
  
  static getDerivedStateFromProps(props, state) {...}  
  
  render() {  
    console.log('render');  
  }  
}
```

```

    return <div>{this.state.time}</div>;
  }
}

```

Listing 5.12 Implementation of the “render” Method (src/Timer.jsx)

The last step in the mounting section involves the `componentDidMount` method.

5.5.4 “componentDidMount”

With the methods used so far, you shouldn’t trigger any side effects yet as they can be aborted by React, which in turn can cause inconsistencies. You can trigger side effects using the `componentDidMount` method. Typically, these consist of asynchronous requests to a server or, as in our current example, setting an interval:

```

import React from 'react';

export default class Timer extends React.Component {
  interval = null;

  constructor(props) {...}

  static getDerivedStateFromProps(props, state) {...}

  render() {...}

  componentDidMount() {
    console.log('componentDidMount');
    this.interval = setInterval(
      () => this.setState(state => ({ time: state.time + 1 })),
      1000,
    );
  }
}

```

Listing 5.13 Implementation of the “componentDidMount” Method (src/Timer.jsx)

In [Listing 5.13](#), you output the information about the execution of the method on the console again. Then you create an interval that increments the `time` value of the state by one every second. After you start your application on the console, switch to the browser. There you should see a result like the one shown in [Figure 5.2](#).

Once the `componentDidMount` method has been run, the component moves on directly to the updating section of its lifecycle. The `getDerivedStateFromProps` and `render` methods will continue to be executed on every update. These updates are caused by the `setState` calls in the `componentDidMount` method.

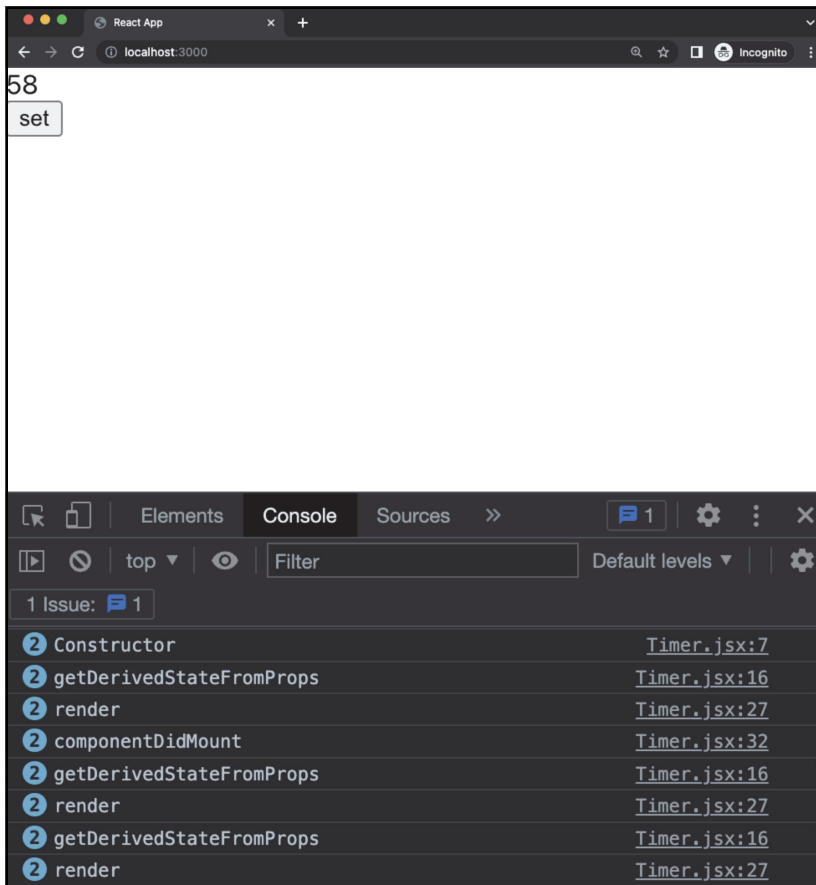


Figure 5.2 Display of the Lifecycle in the Browser

5.5.5 “shouldComponentUpdate”

You can use the `shouldComponentUpdate` method to specify whether the component should be redrawn after a `setState` call. If this method returns `false`, the `render` method won't be executed. The default value `true` ensures that the component will be displayed anew. As arguments, you get access to the current props and the new state. In the example, you make sure that only time entries with an even value are displayed:

```
export default class Timer extends React.Component {
  interval = null;
  constructor(props) {...}
  static getDerivedStateFromProps(props, state) {...}
  render() {... }
  componentDidMount() {...}

  shouldComponentUpdate(newProps, newState) {
    console.log('shouldComponentUpdate');
  }
}
```

```

    return newState.time % 2 === 0;
  }
}

```

Listing 5.14 Implementation of the “shouldComponentUpdate” Method (src/Timer.jsx)

When you return to the browser after this change, you’ll see in the console that the `render` method is only executed on every other `setState`. The `getDerivedStateFromProps` and `shouldComponentUpdate` methods are still called every second.

5.5.6 “getSnapshotBeforeUpdate”

The `getSnapshotBeforeUpdate` method enables you to implement a hook that is executed after the `render` method has been run and before the changes are displayed. In this method, you have access to the previous state and props. The return value of this method must be either `null` or a value. This is then available in the `componentDidUpdate` method. The `getSnapshotBeforeUpdate` method isn’t executed if the `shouldComponentUpdate` method returns `false`.

```

export default class Timer extends React.Component {
  interval = null;
  constructor(props) {...}
  static getDerivedStateFromProps(props, state) {...}
  render() {...}
  componentDidMount() {...}
  shouldComponentUpdate(newProps, newState) {...}

  getSnapshotBeforeUpdate(oldProps, oldState) {
    console.log('getSnapshotBeforeUpdate');
    return Date.now();
  }
}

```

Listing 5.15 Implementation of the “getSnapshotBeforeUpdate” Method (src/Timer.jsx)

In [Listing 5.15](#), you merely generate the current timestamp and return it. You can then access this value in the `componentDidUpdate` method. This also marks the end of an updating section.

5.5.7 “componentDidUpdate”

In the `componentDidUpdate` method, as in the `componentDidMount` method, you can place side effects. The values of the previous props and the previous state are available as arguments in this method. You can use them to find out if certain values have changed in the current updating section. For this purpose, you can access the current values of the props and the state, respectively, via the `props` and `state` properties:

```

export default class Timer extends React.Component {
  interval = null;
  constructor(props) {...}
  static getDerivedStateFromProps(props, state) {...}
  render() {...}
  componentDidMount() {...}
  shouldComponentUpdate(newProps, newState) {...}
  getSnapshotBeforeUpdate(oldProps, oldState) {...}

  componentDidUpdate(oldProps, oldState, snapshot) {
    console.log('componentDidUpdate');
    if (oldState.initial !== this.state.initial) {
      console.log(`${snapshot}: Time was reset`);
    }
  }
}

```

Listing 5.16 Implementation of the “componentDidUpdate” Hook

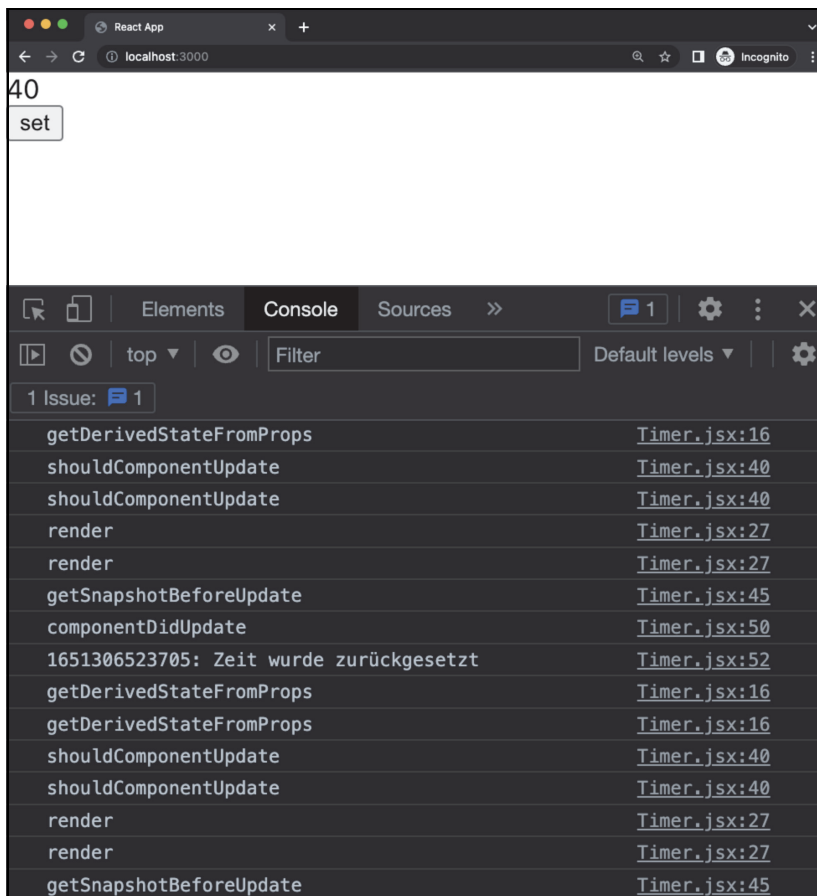


Figure 5.3 Updating Section of the Component

[Listing 5.16](#) shows a concrete example of the `componentDidUpdate` method. Here you check if the `initial` property of the state has changed. If that's the case—that is, the button has been clicked on—then the browser console will show that the time has been reset. It also outputs the timestamp you created earlier in the `getSnapshotBeforeUpdate` method.

[Figure 5.3](#) shows the console output during the updating section—in this case, also with the output after the users have clicked on the button.

The last stage of a component's lifecycle is initiated when the component is removed.

5.5.8 “componentWillUnmount”

A component can be removed by using conditional rendering, which allows you to display a component only under certain conditions. If the condition is no longer true, the component is removed. However, if you allocate resources within a component, you must ensure that these resources will be released again when you unmount the component. A classic example of such a resource is an interval, like the one you start in the timer component. This kind of cleanup work is carried out by the `componentWillUnmount` method.

The `componentWillUnmount` method is called directly before removing a component and receives no arguments. Inside the method, you shouldn't call the `setState` method again because the component won't be rendered again after the execution of the `componentWillUnmount` method.

To demonstrate the `componentWillUnmount` method, you can add an additional button in the `App` component that alternately sets the `show` property of the component state to `true` or `false`. You should set the default value of this property to `true` in the component. Depending on the value of the `show` property, you show or hide the `timer` component. You can achieve this, for example, by using the `&&` operator, as shown in [Listing 5.17](#):

```
class App extends Component {
  state = {
    time: 0,
    show: true,
  };

  getClickHandler() {...}

  handleToggleShow() {
    this.setState((state) => ({ ...state, show: !state.show }));
  }

  render() {
```

```
    return (
      <div className="App">
        {this.state.show && <Timer time={this.state.time} />}
        <button onClick={() => this.handleClick()}>set</button>
        <button onClick={() => this.handleToggleShow()}>toggle</button>
      </div>
    );
  }
}
```

Listing 5.17 Conditional Rendering of the “timer” Component (src/App.jsx)

In the timer component, you implement the `componentWillUnmount` method and use the `clearInterval` function to end the interval. You can see the required source code in [Listing 5.18](#):

```
export default class Timer extends React.Component {
  interval = null;
  constructor(props) {...}
  static getDerivedStateFromProps(props, state) {...}
  render() {...}
  componentDidMount() {...}
  shouldComponentUpdate(newProps, newState) {...}
  getSnapshotBeforeUpdate(oldProps, oldState) {...}
  componentDidUpdate(oldProps, oldState, snapshot) {...}

  componentWillUnmount() {
    console.log('componentWillUnmount');
    clearInterval(this.interval);
  }
}
```

Listing 5.18 Implementation of the “componentWillUnmount” Method (src/Timer.jsx)

5.5.9 Unsafe Hooks

The hook methods presented here have been available in this form only since version 16.3. Prior to that release, there were other methods that are now marked as deprecated and will be removed in future releases. With the deprecated methods, some new features cannot be implemented—such as asynchronous rendering, for example. For this reason, the new lifecycle was introduced.

The methods that will be removed are as follows:

- **componentWillMount**

This method is executed in the mounting section before the `render` method.

- **componentWillReceiveProps**

The `componentWillReceiveProps` method is called before the props are updated. The argument provides access to the new props.

- **componentWillUpdate**

This method is executed once the props and state have been updated and before the component is rendered.

These lifecycle methods are given the prefix `UNSAFE_` to indicate that problems may occur when they are used. The existing components can be automatically updated using the `rename-unsafe-lifecycles` codemod. For more information on this and other codemods, visit <https://github.com/reactjs/react-codemod>.

The different lifecycle methods are used in React applications. A typical example is communication with a server to load the data for display.

5.6 Error Boundaries

In addition to the lifecycle methods presented so far, there are two other methods that have less to do with the lifecycle of a component directly and more with the handling of lifecycle errors and with the `render` method. Usually, you should provide an appropriate error-handling routine within your application at the points where errors may occur.

Uncaught errors cause the entire component tree to be removed. This means that you must take care of error handling in any case. The `getDerivedStateFromError` and `componentDidCatch` methods provide a means for general error handling. If a class component implements one method or both, it's referred to as an *error boundary*. The name refers to the fact that within the `render` method of a component, other components can be included and these themselves can cause errors. Thus, an error boundary takes care of not only its own errors, but also the errors of its child components.

```
import React from 'react';

class ErrorComponent extends React.Component {
  render() {
    throw new Error('oh no!');
    return <h1>Error Component works!</h1>;
  }
}

export default ErrorComponent;
```

Listing 5.19 Component that Throws an Exception in the “render” Method
(src/ErrorBoundary.jsx)

If you include the `ErrorComponent` component from [Listing 5.19](#) in the `App` component of your application, you'll only see a blank page and an error message in the console, as shown in [Figure 5.4](#). The following sections describe how you can deal with such faulty components.

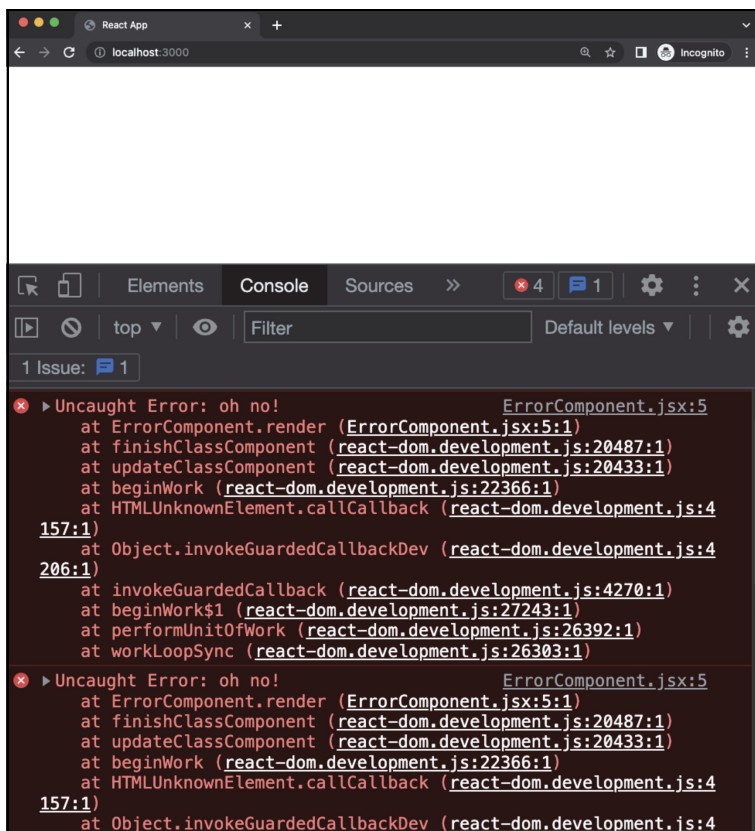


Figure 5.4 Output of a Faulty Component

5.6.1 Logging Errors Using “`componentDidCatch`”

The `componentDidCatch` method is not well-suited for error handling, but rather for logging errors. This method is implemented in a class method and receives two parameters. The first one provides access to the thrown error in the form of an object. The second parameter is a so-called component stack trace. This parameter indicates where exactly in the component tree the error occurred. In [Listing 5.20](#), you can see the source code of the `App` component that contains the `ErrorComponent` component. In addition, the `App` component implements the `componentDidCatch` method:

```
import React from 'react';
import './App.css';
import ErrorComponent from './ErrorComponent';
```

```

class App extends React.Component {
  componentDidCatch(error, info) {
    console.log('*'.repeat(20));
    console.log(error, info);
    console.log('*'.repeat(20));
  }

  render() {
    return <ErrorComponent />;
  }
}

export default App;

```

Listing 5.20 Implementation of an Error Boundary (src/App.tsx)

Figure 5.5 contains the output of the `componentDidCatch` method.

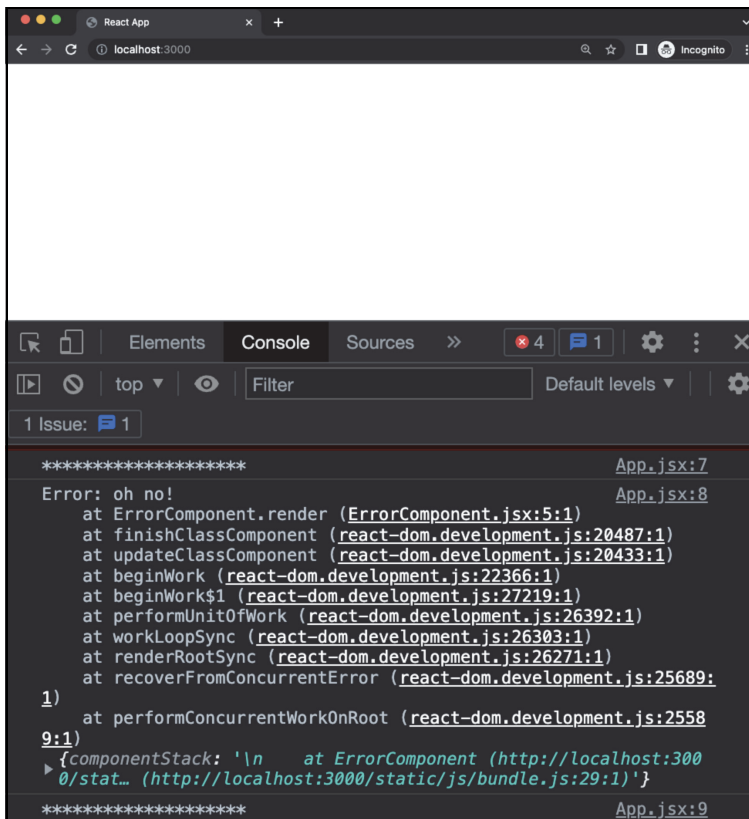


Figure 5.5 Output of the “`componentDidCatch`” Method

The `componentDidCatch` method is executed in the commit phase of the component. This means that side effects are allowed, such as communicating with a server to store the information. As mentioned previously, the `componentDidCatch` method is only used for logging errors. This means that the application won't catch the thrown error, which will cause the application to terminate.

5.6.2 Alternative Representation in Case of an Error with “`getDerivedStateFromError`”

You can use the `getDerivedStateFromError` method to catch errors and make sure that your application continues to function and produce meaningful output even if an error occurs. This static method allows you to set the state of the component. Within this method, you can access the error via the parameter. In [Listing 5.21](#), you make sure that the application displays an appropriate message instead of the component tree to the user in case of an error:

```
import React from 'react';
import './App.css';
import ErrorComponent from './ErrorComponent';

class App extends React.Component {
  state = {
    error: null,
  };

  static getDerivedStateFromError(error) {
    return {
      error: error.message,
    };
  }

  componentDidCatch(error, info) {
    console.log('*'.repeat(20));
    console.log(error, info);
    console.log('*'.repeat(20));
  }

  render() {
    if (this.state.error) {
      return <div>An error has occurred:
        {this.state.error}</div>;
    } else {
      return <ErrorComponent />;
    }
  }
}
```

```

    }
  }
}

```

```
export default App;
```

Listing 5.21 Use of the “`getDerivedStateFromError`” Method (`src/App.jsx`)

The `getDerivedStateFromError` method returns a new state object for the component. Due to this state, you can display the error message inside the render method. In addition to this method, you can also use the `componentDidCatch` method to log the error message.

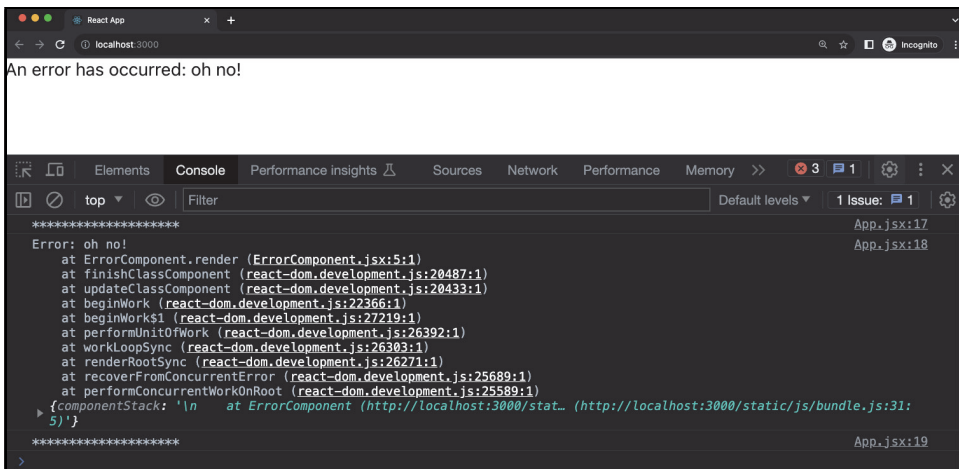


Figure 5.6 Catching the Error Using the “`getDerivedStateFromError`” Method

Based on the features presented so far, you now know almost everything you need to know about class components. The last important building block of a React application you can use directly in your class component is the Context API.

5.7 Using the Context API in a Class Component

The Context API allows you to access centralized information, independent of the props. For this purpose, you must first create a context. In [Listing 5.22](#), you can see how to create the context using the `createContext` function:

```
import { createContext } from 'react';

export const BooksContext = createContext([]);
```

Listing 5.22 Creating “BooksContext” (`src/BooksContext.jsx`)

You include this context in the `App` component of your application and place a data record in the context. The corresponding source code of the `App` component is shown in [Listing 5.23](#):

```
import React from 'react';
import './App.css';
import { BooksContext } from './BooksContext';
import BooksList from './BooksList';

class App extends React.Component {
  state = {
    books: [
      {
        id: 1,
        title: 'JavaScript - The Comprehensive Guide',
        author: 'Philip Ackermann',
        isbn: '978-3836286299',
        rating: 5,
      },
    ],
  };

  render() {
    return (
      <BooksContext.Provider value={this.state.books}>
        <BooksList />
      </BooksContext.Provider>
    );
  }
}

export default App;
```

Listing 5.23 Integrating the Context into the Application (`src/App.jsx`)

In the `App` component, you integrate `BooksContext.Provider` and assign it the state with the data array as a value. This allows all child components, such as the `BooksList` component, to access it. You can see the source code of the `BooksList` component with context access in [Listing 5.24](#):

```
import React from 'react';
import { BooksContext } from './BooksContext';

class BooksList extends React.Component {
  static contextType = BooksContext;
```

```
render() {  
  return (  
    <ul>  
      {this.context.map((book) => (  
        <li key={book.id}>{book.title}</li>  
      ))}  
    </ul>  
  );  
}
```

```
export default BooksList;
```

Listing 5.24 Accessing the Context from within a Class Component (src/BooksList.jsx)

In a class component, you first define a static property named `contextType` for accessing the context. To this property, you assign the `BooksContext` object. This configures the component so that React gives you access to the context directly through the `context` property of the class.

5.8 Differences between Function and Class Components

In a modern React application, you'll no longer find class components, and there are good reasons for this. One of the most obvious differences is the syntax and the overhead that comes with it. Function components are more lightweight. In the simplest case, you define a function and return a JSX structure. You don't need to import React.

The difference between function and class components becomes really clear when it comes to state and lifecycle.

5.8.1 State

In a class component, you have a defined `state` property and the `setState` method to set the state. Changing the state will cause React to render the component anew.

Function components work similarly. You use the `useState` function, which returns an array for reading and writing the state. The big difference between the two approaches is that you can define multiple state fragments in a function component. This gives you more flexibility and allows you to name the state appropriately, making the source code of your application more understandable.

5.8.2 Lifecycle

Things look similar for the lifecycle. In a class component, you have several methods that allow you to intervene in the lifecycle of a component in a very fine-grained manner.

For function components, you use the `useEffect` function and can respond to mounting, updating, and unmounting a component. The dependency array and cleanup routines give you additional flexibility.

Unlike the lifecycle methods of the class component, one of which is available to you for each phase of the lifecycle, you can define several specialized lifecycle functions via the `useEffect` function. This also makes your source code more flexible, readable, and maintainable.

5.9 Summary

In this chapter, you learned about the class components of React. Although you should avoid these types of components in general, you cannot always do so, especially in older legacy applications, and therefore it's worth knowing the peculiarities of class components:

- Class components are JavaScript classes that derive from the `React.Component` class and have at least one `render` method.
- Like function components, class components accept the passing of information from the parent component in the form of attributes called *props*. You can access the values of the props via `this.props`.
- Class components can have their own state. You have read access to it via `this.state` and write access via the `setState` method. A change causes React to re-render the component.
- Using various lifecycle methods, you can respond to the phases of the component's lifecycle. One of the most important methods is `componentDidMount`, which is executed by React once the component has been mounted. In this method, you can trigger side effects, such as communication with a web server.
- You can use a combination of the static `contentType` property and the `context` property of the component class to access the Context API of React.
- You can use error boundaries for logging or catching errors.
- Function components are more flexible and lightweight compared to class components. They also allow multiple small state fragments and multiple lifecycle functions.

In the next chapter, you'll learn more about the Hooks API of React, the interface that led to the replacement of class components.

Chapter 6

The Hooks API of React

Version 16.8 introduced a comparatively extensive extension to React: the Hooks API. For many React developers, the Hooks API marks a turning point in writing React applications. Some consider the Hooks API to be so important that there is even talk of moving from ReactJS to React, as was the case with Angular during the transition from version 1 to version 2. However, Angular has been rewritten from the ground up. In contrast, the Hooks API is merely an extension. The foundation for this strategic expansion of the library was laid by the integration of the new Fiber reconciler.

At this point, the following question arises: If the Hooks API is such a significant enhancement, why hasn't the version number been increased to 17? React follows the semantic versioning approach, which states that a breaking change leads to an increase in the major version. However, such a change isn't present in the Hooks API. Indeed, you can implement your application completely without hooks and with no fear of any restrictions.

The Hooks API represents an upgrade to the functional components. Prior to the introduction of the new interface, it was only possible to implement a local state and access lifecycle methods in class components. These and other restrictions have been largely removed by the hooks. The reasons for introducing hooks are primarily the following:

- **Reduction of the component scope**

Often, it isn't possible without a problem to swap out the state and the associated methods of a component and thus reduce the complexity with the existing means. Due to the Hooks API, it becomes possible to split an extensive state into several substates and manage them separately.

- **Removal of duplicates**

Previously, the state was tightly bound to a class component. It wasn't possible to reuse the structure and methods for manipulating the state in multiple places. You can also find a solution for this with the Hooks API.

- **Replacement of complex design patterns**

With higher-order components and render props, you've gotten to know two design patterns that help you extend the functionality of components. Custom hooks provide an alternative in this context.

Throughout this chapter, you'll learn about the benefits and architectural approaches of the Hooks API and how you can use it in your application. By using hooks, you can

swap out the logic from a component to smaller units. React applications have always been characterized by a clearly modeled data flow and the composition of components. You can pursue these design patterns not only between the individual components of your application, but also within a component, thanks to the Hooks API.

The advantage of the Hooks API in conjunction with function components over class components is that you can implement a local and self-contained state through the Hooks API. The introduction of hooks is intended to significantly lower the barrier to entry. In addition, the features to be learned should be reduced somewhat in the long term. Function components are generally easier to understand and do not have problems related to context—that is, the `this` within a component.

6.1 A First Overview

What you'll quickly notice when dealing with hooks is the naming convention: Hook functions always start with the word `use`, followed by the actual name of the hook. The name of the state hook is therefore `useState`. Hooks are divided into two categories: basic hooks and additional hooks.

6.1.1 The Three Basic Hooks

You already know about the three most important hooks of the Hooks API from the previous chapters. They are grouped together in the *basic hooks* category:

- **`useState`**

The *state hook* makes sure that you can implement a local state within a function component. This state survives several `render` cycles.

- **`useEffect`**

The *effect hook* can be used to emulate some lifecycle methods. This hook allows you to respond to the mounting of a component, changes to a component, and the unmounting of a component.

- **`useContext`**

The *context hook* provides convenient access to the Context API of React. In the context, you can store objects in a central location and access them from elsewhere.

In addition to these three basic hooks, there are several other hooks that can be useful when implementing an application.

6.1.2 Other Components of the Hooks API

The three basic hooks are often referred to in discussions about hooks in the React context. However, it's also worth looking at the other hooks provided by React. The individual hooks are as follows:

- **useReducer**

The *reducer hook* serves a similar purpose as the state hook: it provides a local state in a function component. The difference between the two is that with the reducer hook, you use the Flux architecture to modify the state.

- **useCallback**

This hook returns a memoized callback function that can be used for performance optimization.

- **useMemo**

Like the callback hook, the *memo hook* is used to improve performance and returns a memoized value.

- **useRef**

You can use the *ref hook* to control access to JSX elements. *Refs* denote references to HTML elements in the React environment.

- **useImperativeHandle**

This rather rarely used hook is used to customize the interface of a component when it's used as a ref.

- **useLayoutEffect**

The *layout effect hook* is similar to the effect hook for handling side effects and life-cycle functions. The layout effect hook has the special feature that the side effects are executed synchronously after all DOM changes.

- **useDebugValue**

This hook is used to generate output on the developer tools.

- **useDeferredValue**

Using the *deferred value hook*, you can add a delay. In the case of time-consuming operations, you sometimes have to introduce an artificial delay. React takes care of that for you with this hook. However, the delay depends on the rendering process and not on external factors.

- **useTransition**

This hook allows you to mark an interface update as noncritical so that React can decide which update to give preference to.

- **useId**

This hook creates unique IDs that you can use in various places in your components.

In addition to the extended Hooks API, there are two hook functions aimed at library authors:

- **useSyncExternalStore**

You can use this hook to connect external data sources.

- **useInsertionEffect**

This hook is similar to the other two effect hooks. However, React executes it before any DOM changes.

Memoization

In software development, the term *memoization* refers to the caching of a value. For example, if you memoize a callback function, the return value is cached and returned directly when called again, rather than being recalculated. Memoization can result in a significant performance improvement, especially when you consider that React's render functions may be run very frequently. If you define function objects within such a render function, such as for event handlers, this means that these function objects must be generated anew for each render process.

You already know the three basic hooks from the previous chapters. In [Chapter 3](#), you saw how to use `useState` in your function components. In [Chapter 4](#), you looked at the use of `useEffect` and `useContext`, among other things.

In this chapter, we'll look at the other hook features React makes available to you. The reducer hook will start us off.

6.2 “useReducer”: The Reducer Hook

Quick Start

```
const [state, dispatch] = useReducer(state, reducer);
```

Listing 6.1 Syntax of the Reducer Hook

The reducer hook works like the state hook, with the difference that you do not manipulate the state directly via a setter function but use the `dispatch` function to dispatch action objects. You process the action objects in the `reducer` function and create a new state on this basis.

You can use the reducer hook as a replacement for the state hook. [Listing 6.2](#) shows an implementation of the `BooksList` component that creates a local state via the `useReducer` function. The only operation allowed by this component is to evaluate the entries.

```
import { useReducer } from 'react';
import produce from 'immer';
import { StarBorder, Star } from '@mui/icons-material';

function reducer(state, action) {
  switch (action.type) {
    case 'RATE':
      return produce(state, (draftState) => {
        const index = draftState.findIndex(
```

```

        (book) => book.id === action.payload.id
      );
      draftState[index].rating = action.payload.rating;
    });
    default:
      return state;
  }
}

const initialBooks = [
  {
    id: 1,
    title: 'JavaScript - The Comprehensive Guide',
    author: 'Philip Ackermann',
    isbn: '978-3836286299',
    rating: 5,
  },
  ...
];

function BooksList() {
  const [books, dispatch] = useReducer(reducer, initialBooks);
  return (
    <table>
      <thead>
        <tr>
          <th>Title</th>
          <th>Author</th>
          <th>ISBN</th>
          <th></th>
        </tr>
      </thead>
      <tbody>
        {books.map((book) => (
          <tr key={book.id}>
            <td>{book.title}</td>
            <td>{book.author}</td>
            <td>{book.isbn}</td>
            <td>
              {new Array(5).fill('').map((item, i) => (
                <button
                  className="ratingButton"
                  key={i}
                  onClick={() =>

```

```
        dispatch({
          type: 'RATE',
          payload: { id: book.id, rating: i + 1 },
        })
      }
    >
    {book.rating < i + 1 ? <StarBorder /> : <Star />}
  </button>
))}
</td>
</tr>
))}
</tbody>
</table>
);
}
```

```
export default BooksList;
```

Listing 6.2 Using the Reducer Hook (src/BooksList.jsx)

The core of this component, which renders a list of books, is the call of the `useReducer` function. When calling it, you pass a reducer function and an initial state to it. As a return, you get an array with two elements. The first element provides read access to the state, similar to the state hook. The second element of the array is the `dispatch` function.

6.2.1 The Reducer Function

The reducer function is the eponymous element of the reducer hook. Like the setter function of the state, it is a function that gets the previous state as well as an action object. The action object describes the change to be made to the state. The core of the reducer function is a `switch` statement in which you decide to what branch you want to jump based on the `type` property of the action object. In the case of the `RATE` type, you use `Immer` to create a copy of the state and set the new rating for the specified record. Then you return the modified state as the value of the reducer function.

6.2.2 Actions and Dispatching

The second element that makes up the reducer hook is an action. *Actions* are simple JavaScript objects that describe a change to the state. There is a best practice that such an action object should have a `type` property as well as a `payload` property. You use the `type` property to decide which operation you want to perform in the reducer function. The `payload` property can have any data structure required to control the operation. In

the case of the rating example, the `payload` property contains an object with the ID of the affected record and the new rating value.

What’s still missing is the connection between the action object and the reducer function. This is where the `dispatch` function comes into play. You get this function as the second element of the return value of the `useReducer` function. The `dispatch` function makes sure that the reducer function is executed along with the action object that has been passed in order to produce a new state and re-render the component. In the example, you call the `dispatch` function in the click handler of the rating buttons.

The rating example contains only one operation, and it is synchronous—for good reason. It isn’t possible to handle asynchronicity within the reducer function. In the next section, you’ll learn how to deal with asynchronous operations.

6.2.3 Asynchronicity in the Reducer Hook

You have several options to deal with asynchronicity:

- The first and simplest option is to first perform the asynchronous operation (e.g., loading or writing data) and then dispatch a corresponding action. However, by doing so, you lose the advantage of the reducer hook, which allows you to pull operations out of your component and encapsulate them cleanly.
- The second variant is to attach one or more effect hooks to your reducer hook, which respond to the corresponding state changes and encapsulate the asynchronous side effects. However, this variant makes the source code significantly more complex, and the dependencies between the state and the effect hooks are not necessarily always directly obvious.
- The third and most elegant variant is that you create an additional wrapper function to wrap the reducer hook, handle the asynchronicity, and dispatch the action. [Listing 6.5](#) shows the implementation of such a middleware option that allows for loading and evaluating data records with a server.

Due to the asynchronous middleware, this example is a bit more extensive and is divided into three parts: middleware function, reducer function, and the `BooksList` component. For a better understanding, let’s look at the three parts separately. To test the code yourself, you’ll want to put the individual parts together in one file.

```
import { useReducer, useMemo, useEffect } from 'react';
import produce from 'immer';
import { StarBorder, Star } from '@mui/icons-material';

function middleware(dispatch) {
  return async function (action) {
    // eslint-disable-next-line default-case
    switch (action.type) {
```

```
case 'FETCH':
  const fetchResponse = await fetch(
    `${process.env.REACT_APP_API_SERVER}/books`
  );
  const books = await fetchResponse.json();
  dispatch({ type: 'LOAD_SUCCESS', payload: books });
  break;
case 'RATE':
  await fetch(
    `${process.env.REACT_APP_API_SERVER}/books/${action.payload.id}`,
    {
      method: 'PUT',
      headers: { 'Content-Type': 'Application/JSON' },
      body: JSON.stringify(action.payload),
    }
  );
  dispatch({
    type: 'RATE_SUCCESS',
    payload: { id: action.payload.id
rating:action.payload.rating },
  });
  break;
}
};
}
```

Listing 6.3 Middleware Function (src/Bookslist.jsx)

The middleware function accepts a reference to the `dispatch` function you get from the reducer hook as a return value as an argument, and in turn returns a function itself. This function assumes the role of the `dispatch` function and is called with an action object. The function body consists of a `switch` statement that decides what needs to be done based on the action passed. In the example, you implement cases for actions using the `FETCH` and `RATE` types. In the case of `FETCH`, you load the data for the display from the server interface and then execute the `dispatch` function with a new action object. This is of type `FETCH_SUCCESS` and contains the loaded data as payload. The action is then processed by the actual reducer function.

The second action supported by the middleware is of the `RATE` type. In this case, you perform an asynchronous write operation on the server. To do this, you use the `PUT` method and configure the request so that it has the correct content type and contains the modified record as the payload. If the server reports a success of the write operation, you dispatch a `RATE_SUCCESS` action, which in turn is further processed by the reducer function.

In the next step, you implement the reducer function that takes care of processing the `FETCH_SUCCESS` and `RATE_SUCCESS` actions. [Listing 6.4](#) contains the source code of the function:

```
function reducer(state, action) {
  switch (action.type) {
    case 'LOAD_SUCCESS':
      return action.payload;
    case 'RATE_SUCCESS':
      return produce(state, (draftState) => {
        const index = draftState.findIndex(
          (book) => book.id === action.payload.id
        );
        draftState[index].rating = action.payload.rating;
      });
    default:
      return state;
  }
}
```

Listing 6.4 Reducer Function (src/BooksList.jsx)

As with previous implementations, this reducer function is completely synchronous in design. The middleware function triggers the reducer function by calling the `dispatch` function. In the case of the `FETCH_SUCCESS` action, you return the data of the action object stored in the `payload` property.

If your application triggers the `RATE_SUCCESS` action, you use the Immer library to create a copy of the previous state and change the rating of the affected record.

Using these two functions, you have done all the preparatory work and can move onto implementing the component. This component is responsible for the state, must trigger the loading of the data, and displays the list of books. [Listing 6.5](#) contains the implementation of the `BooksList` component:

```
function BooksList() {
  const [books, dispatch] = useReducer(reducer, []);

  const middlewareDispatch = useMemo(() => middleware(dispatch), [dispatch]);

  useEffect(() => {
    middlewareDispatch({ type: 'FETCH' });
  }, [middlewareDispatch]);

  return (
    <table>
```

```
<thead>
  <tr>
    <th>Title</th>
    <th>Author</th>
    <th>ISBN</th>
    <th></th>
  </tr>
</thead>
<tbody>
  {books.map((book) => (
    <tr key={book.id}>
      <td>{book.title}</td>
      <td>{book.author}</td>
      <td>{book.isbn}</td>
      <td>
        {new Array(5).fill('').map((item, i) => (
          <button
            className="ratingButton"
            key={i}
            onClick={() =>
              middlewareDispatch({
                type: 'RATE',
                payload: { ...book, rating: i + 1 },
              })
            }
          >
            {book.rating < i + 1 ? <StarBorder /> : <Star />}
          </button>
        ))}
      </td>
    </tr>
  ))}
</tbody>
</table>
);
}
```

```
export default BooksList;
```

Listing 6.5 “BooksList” Component with Reducer Hook and Middleware (src/BooksList.jsx)

In the first step, you call the `useReducer` function to get access to the state and dispatch function. As the initial value, you pass an empty array. This results in the component not displaying any data at first. Then you call the `middleware` function and pass the

dispatch function to it. At this point, you use the `useMemo` function that ensures the middleware function will not be recreated for each render operation.

Finally, using the effect hook, you make sure that the data for the display is loaded from the server. You call the `middlewareDispatch` function with a `FETCH` action object, which is first processed by the middleware and then sets the state via the reducer function, leading to the display of the data.

In the table, you represent the individual data records, and when a rating star is clicked on, you make sure that the `RATE` action will be processed by the middleware. Again, the middleware triggers the server communication and then the `RATE_SUCCESS` action, which then leads to the update of the representation via the reducer.

6.3 “useCallback”: Memoizing Functions

Quick Start

You can use the `useCallback` function to create a memoized version of a function that React recreates only when one of the specified dependencies changes.

```
const func = useCallback((params) => { /*logic*/ }, [dependency]);
```

Listing 6.6 Syntax of the “useCallback” Function

[Listing 6.6](#) shows how to use the `useCallback` function to memoize the passed function. If the dependency variable changes, React creates a new version of the function and stores it in the `func` variable.

The callback hook is intended to optimize the rendering process. If you define a function within a component, it will be created again during each render process. In most cases, this is unnecessary. You can use the `useCallback` function to make React reuse the function unless certain aspects of the function change that require the function to be created.

Memoizing functions is especially interesting if you have a lot of functions in your component tree and this becomes a noticeable performance problem during render operations. You shouldn’t try to optimize every callback function in your components using the callback hook as it often happens that the hoped-for effect fails to materialize and the source code becomes less readable. In [Listing 6.7](#), you can see an example of using the callback hook:

```
import { useCallback } from 'react';
import Rating from './Rating';

function BooksListItem({ book, onRate }) {
  const handleRate = useCallback(
```



```

(event) => {
  const rating = event.target.closest(
    '[data-value]')?.dataset.value;
  if (rating) {
    onRate(book.id, parseInt(rating, 10));
  }
},
[book.id, onRate]
);

return (
  <tr>
    <td>{book.title}</td>
    <td>{book.author ? book.author : 'Unknown'}</td>
    <td>{book.isbn}</td>
    <td onClick={handleRate}>
      <Rating item={book} />
    </td>
  </tr>
);
}

```

Listing 6.7 Using the “useCallback” Function (src/BooksListItem.jsx)

The component is the `BooksListItem` component, which is responsible for displaying a table row with a data record. The component has the `rating` component as a child component, which is responsible for displaying the rating and the option to change it. The `BooksListItem` component has a `handleRate` function that receives a `click` event and extracts the rating value from it, which is located in the `data-value` property of the clicked-on HTML element. Usually, this function is regenerated with each render operation.

If you use the `useCallback` function, React regenerates the function only if the ID of the record or the `onRate` function changes. The function needs both to do its job correctly. For this reason, you should name the information in the dependency array. The return value of the `useCallback` function is the memoized original function, which you can use like an ordinary function in your application.

6.4 “useMemo”: Memoizing Objects

Quick Start

You can use the memo hook to create a memoized version of any object structure. The functionality is similar to that of `useCallback`:

```
const memoObject = useMemo(() => doSomething(obj), [obj]);
```

Listing 6.8 Syntax of the “useMemo” Function

In the case of [Listing 6.8](#), React stores the return value of the `doSomething` function in the `memoObject` variable and regenerates it if the `obj` dependency changes.

You’ve already seen the use of the `useMemo` function in this chapter when we talked about the asynchronous middleware in the reducer hook.

```
function reducer(state, action) {...}

function middleware(dispatch) {...}

function BooksList() {
  const [books, dispatch] = useReducer(reducer, []);

  const middlewareDispatch = useMemo(
    () => middleware(dispatch), [dispatch]
  );

  useEffect(() => {
    middlewareDispatch({ type: 'FETCH' });
  }, [middlewareDispatch]);

  ...
  return (
    <table>...</table>
  );
}
```

Listing 6.9 Using the “useMemo” Function (src/BooksList.jsx)

The middleware function from [Listing 6.9](#), to which you pass the `dispatch` function, creates the `middlewareDispatch` function that you can use in your component to trigger actions. You only need to regenerate the `middlewareDispatch` function if the `dispatch` function changes. The `middlewareDispatch` variable then contains a memoized variant of the generated function. You can then use this function in the effect hook, where you can also name it as a dependency as the function doesn’t change between the individual render operations.

If you were to run the middleware function to create the `middlewareDispatch` function without `useMemo`, the result would be that React would recreate the function on every render and thus also rerun the effect hook to load the data.

Note

`useCallback` and `useMemo` are very similar. You can use the `useMemo` function to simulate the same behavior as `useCallback`. `useCallback(func, deps)` is the same as `useMemo(() => func, deps)`.

6.5 “useRef”: References and Immutable Values

Quick Start

The ref hook allows you to create a changeable reference. This will outlast the render operations of your component, similar to state. However, changing the reference doesn't cause the component to be rendered again.

```
const ref = useRef(null);
ref.current = 'myValue';
```

Listing 6.10 Using the Ref Hook

You can pass an initial value for the ref to the `useRef` function. Then you use the value via the `current` property of `ref`, and you have both read and write access to it.

You can use the ref hook in two very different ways. On the one hand, you can use it to manage references to HTML elements, and on the other hand, you can keep values that are not reset by re-rendering the component. Let's first turn our attention to references to HTML elements in the shape of form elements.

6.5.1 Form Handling Using the Ref Hook

React takes a declarative approach to generating graphical interfaces. Direct access to HTML elements is a sign of poor application design. However, there are cases when you need to access the HTML structure. An example of this is a variant of form handling:

```
import { useState, useRef } from 'react';
import './App.css';

function App() {
  const [name, setName] = useState('');
  const inputRef = useRef();

  function handleChange() {
    setName(inputRef.current.value);
  }
}
```

```

    return (
      <div>
        <div>{name}</div>
        <input type="text" ref={inputRef} onChange={handleChange} />
      </div>
    );
  }

export default App;

```

Listing 6.11 Form Handling Using the “useRef” Function (src/App.jsx)

The code in [Listing 6.11](#) shows the state of the component and an input field. In addition to the state, you also define a ref at the beginning, which you store in the `inputRef` variable. In the JSX structure, you use the `ref` prop of the `input` element to connect the ref to the input element. In the change handler of the input element, you can then access the element via `inputRef.current` and use the `value` property to read the current value of the element and write it to the state. React then re-renders the component and displays the new value.

6.5.2 Caching Values Using the Ref Hook

The second use for the ref hook is to store information similar to the state hook, except that changing the ref doesn’t cause the component to be re-rendered. You’ve already learned about using the ref hook in [Chapter 4](#). Here’s another short example with a timer component that creates an interval and deletes it when the component becomes unhooked:

```

import { useState, useEffect, useRef } from 'react';
import './App.css';

function App() {
  const [showTimer, setShowTimer] = useState(true);
  useEffect(() => {
    setTimeout(() => setShowTimer(false), 5000);
  });

  return <div>{showTimer && <Timer />}</div>;
}

export default App;

function Timer() {
  const [time, setTime] = useState(0);
  const intervalRef = useRef(null);
  useEffect(() => {

```

```
    intervalRef.current = setInterval(  
      () => setTime((prevTime) => prevTime + 1),  
      1000  
    );  
    return () => clearInterval(intervalRef.current);  
  }, []);  
  
  return <div>{time}</div>;  
}
```

Listing 6.12 Storing Values in Ref Hook (src/App.jsx)

The example consists of two components: the `App` component and the `timer` component. The `App` component has a Boolean state that determines whether the `timer` component is displayed or not. After five seconds, the value is set to `false` via an effect hook so that the `timer` component is unhooked.

The `timer` component also has its own state, which stores the time in seconds from the point at which the component was mounted. You also create a ref where you store the handle, which refers to the interval. In the effect hook of the component, you start the interval and update the state once every second. In the cleanup routine of the effect hook, you call the `clearInterval` function with the value from the ref. This will ensure that the interval isn't accidentally left open to cause potential memory problems.

6.6 “useImperativeHandle”: Controlling Forward Refs

Quick Start

You can use the `useImperativeHandle` function to replace the object in a *forward ref*. For this purpose, you pass the ref object and a function that replaces it. In addition, you can pass optional dependencies, which, if changed, will cause the function to be executed again and thus regenerate the object.

```
useImperativeHandle(ref, createHandle, [deps])
```

Listing 6.13 Syntax of the “useImperativeHandle” Function

The `useImperativeHandle` function gives you better control over forward refs. To that end, let's first look at this feature of React before looking at the actual hook function.

6.6.1 Forward Refs

The concept of forward refs refers to a pattern where you pass a ref to an element from a child component to a parent component. To make this work, you use the `forwardRef`

function and pass to it the child component that contains the element to be referenced. In [Listing 6.14](#), you can see the child component:

```
import { forwardRef } from 'react';

function Input({ label }, ref) {
  return (
    <div>
      <label>{label}</label>
      <input type="text" ref={ref} />
    </div>
  );
}

export default forwardRef(Input);
```

Listing 6.14 Child Component with “forwardRef” (src/Input.jsx)

The Input component from the example accepts a label prop as the label of the input field, and via the forwardRef function that surrounds it, the component function receives the forward ref as the second argument. You then associate this with the ref attribute of the input element.

In the next step, you’ll see in [Listing 6.15](#) how you can access the ref of the input element from the parent component:

```
import { useState, useRef } from 'react';
import Input from './Input';
import './App.css';

function App() {
  const ref = useRef(null);

  const [name, setName] = useState('');

  function handleClick() {
    setName(ref.current.value);
  }

  return (
    <div>
      <div>Hello {name}!</div>
      <Input title="Name: " ref={ref} />
      <button onClick={handleClick}>say hello</button>
    </div>
  );
}
```

```
    );  
  }  
  export default App;
```

Listing 6.15 Integration of the “Input” Component via “ForwardRef” (src/App.jsx)

In the parent component, the App component in this case, you first define a ref and then the state you want to use for the display. In the JSX structure, you show the state, and you also include the Input component and pass it the title prop and the ref via the ref prop. Finally, you define a button element to whose click event you bind a function that reads the value from the ref and writes it to the state.

This will pass the reference to the input element from the Input component to the parent component, and you can access it directly from there.

6.6.2 The Imperative Handle Hook

You can use the ImperativeHandle function to further influence the behavior of the forward ref by overriding the ref object. In [Listing 6.16](#), you’ll first see the implementation of the Input component:

```
import { useRef, forwardRef, useImperativeHandle } from 'react';  
  
function Input({ label }, ref) {  
  const inputRef = useRef(null);  
  useImperativeHandle(ref, () => ({  
    getUCValue() {  
      return inputRef.current.value.toUpperCase();  
    },  
  }));  
  
  return (  
    <div>  
      <label>{label}</label>  
      <input type="text" ref={inputRef} />  
    </div>  
  );  
}  
  
export default forwardRef(Input);
```

Listing 6.16 Using the Imperative Handle Hook (src/Input.jsx)

In the child component, you first define your own ref to access the input element. Then you call the useImperativeHandle function with the ref passed to the component and a

function. This function creates an object with a `getUCValue` function that returns the current value of the `input` element and converts it to uppercase.

Due to the use of the `useImperativeHandle` function, you can no longer directly access the `value` property of the `input` element in the parent component. However, you can run the `getUCValue` function instead, as shown in [Listing 6.17](#):

```
import { useState, useRef } from 'react';
import Input from './Input';
import './App.css';

function App() {
  const ref = useRef(null);

  const [name, setName] = useState('');

  function handleClick() {
    setName(ref.current.getUCValue());
  }

  return (
    <div>
      <div>Hello {name}!</div>
      <Input title="Name: " ref={ref} />
      <button onClick={handleClick}>say hello</button>
    </div>
  );
}
export default App;
```

Listing 6.17 Integration of the “Input” Component via the “ImperativeHandle” Hook in the Parent Component (`src/App.jsx`)

6.7 “useLayoutEffect”: The Synchronous Alternative to “useEffect”

Quick Start

The `useLayoutEffect` function works exactly like the `useEffect` function, but with the difference that React executes `useEffect` asynchronously and `useLayoutEffect` synchronously.

The `useLayoutEffect` function is one of the hook functions you’ll come across quite rarely. As a rule, you’ll use `useEffect` and do well with it. The difference between the two functions is the time of execution. In the effect hook, React renders a component. Then

the rendering is updated, and only then does React execute the effect hook asynchronously. So the effect hook doesn't block the display.

With the *layout effect hook*, React also renders the component, but then executes the layout effect—and only after this is complete does React update the rendering.

One situation where you should use the layout effect hook instead of the effect hook is when your component flickers during rendering—that is, it renders first and then immediately renders again.

6.8 “useDebugValue”: Debugging Information in React Developer Tools

To use the debug value hook, you must have React Developer Tools installed and enabled in your browser. These are available for all Chrome-based browsers as well as Firefox. For this hook, I'm getting a little ahead of myself. You can use the debug value hook in conjunction with *custom hooks*, which you'll learn about later in this chapter. The idea is to swap out React hooks into separate functions, which in turn can be used in components. [Listing 6.18](#) contains an example of a custom hook function. This function swaps out the state as well as the effect of a component into a function named `useName`:

```
import { useState, useEffect, useDebugValue } from 'react';
import './App.css';

function useName() {
  const [name, setName] = useState('');
  useDebugValue(`Name is ${name}`);
  useEffect(() => {
    setTimeout(() => {
      setName('React');
    }, 5000);
  });

  return name;
}

function App() {
  const name = useName();

  return <div>{name}</div>;
}

export default App;
```

Listing 6.18 Using “useDebugValue” in a Custom Hook (src/App.jsx)

You can use the `useDebugValue` function to create additional labels and debugging output in React Developer Tools. In this case, you output the value of the `name` state. The dev tools show you the string directly and update it if the value changes. To generate the output as shown in [Figure 6.1](#), go to the **Components** tab of your browser's dev tools and select the **App** component there. By default, the **hooks** section then shows you only the name of the hook—that is, `Name`. However, via the debug value hook, you will then see the additional debugging output, which may include, for example, certain properties of state objects or the like.

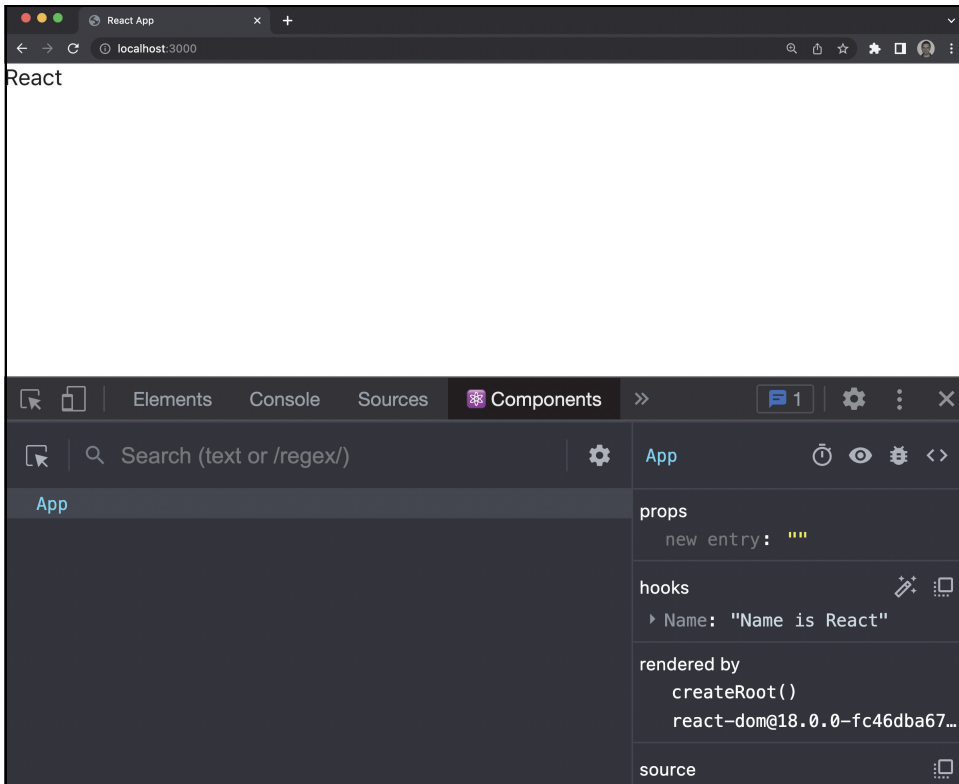


Figure 6.1 Output of the Debug Value Hook in React Developer Tools

With React 18, some additional hook features have made their way into the core of React. In the following sections, you’ll learn more about these.

6.9 “useDeferredValue”: Performing Updates According to Priority

Quick Start

You pass a state value and a timeout in milliseconds to the `useDeferredValue` function. React then uses the previous value up to a maximum of this timeout value. This gives

the library the opportunity to process changes with higher priority and only then take care of the less important update.

```
const deferredValue = useDeferredValue(name, {timeoutMs: 1000});
```

Listing 6.19 Syntax of the “useDeferredValue” Function

With React 16, numerous changes to the core of React were introduced. Some of these changes appeared directly as new features. Other customizations have only gradually found their way into applications. An important feature of React is that the library recognizes different priorities of changes and that it's possible to pause, resume, or restart the render process. Only a few applications and libraries have made use of this feature so far. However, the `useDeferredValue` function now enables you to intervene directly in the rendering process.

In the following example, you'll implement the display of a book list, where you have the option to filter the entries. For the example to work, you need a JSON file with data as shown in [Listing 6.20](#). You save this file as `data.json` in the root directory of your application.

```
{
  "books": [
    {
      "id": 1,
      "title": "JavaScript - The Comprehensive Guide",
      "author": "Philip Ackermann",
      "isbn": "978-3836286299",
      "rating": 5
    },
    {
      "id": 2,
      "title": "Clean Code",
      "author": "Robert Martin",
      "isbn": "978-0132350884",
      "rating": 4
    },
    {
      "id": 3,
      "title": "Design Patterns",
      "author": "Erich Gamma",
      "isbn": "978-0201633610",
      "rating": 5
    }
  ]
}
```

Listing 6.20 Data for the Server Interface (`data.json`)

You can then start the server process from the command line using the `npx json-server -p 3001 -w data.json` command. Then you can implement the `App` component, which contains the search form and integrates the `BooksList` component that takes care of displaying the list. The source code of the `App` component is shown in [Listing 6.21](#):

```
import { useState, useDeferredValue, useMemo } from 'react';
import BooksList from './BooksList';
import './App.css';

function App() {
  console.log('render');
  const [searchString, setSearchString] = useState('');
  const deferredSearchString = useDeferredValue(searchString, {
    timeoutMs: 1000,
  });

  const list = useMemo(() => {
    console.log('memo');
    return <BooksList searchString={deferredSearchString} />;
  }, [deferredSearchString]);

  return (
    <div>
      <div>
        Search: { ' '}
        <input
          type="text"
          value={searchString}
          onChange={(event) => {
            setSearchString(event.target.value);
          }}
        />
        {list}
      </div>
    </div>
  );
}

export default App;
```

Listing 6.21 Implementation of the Search Form (src/App.jsx)

In the `App` component, you connect the `input` element to the `searchString` state by assigning the value of the state to the `value` prop of the element and using a change handler to manipulate the state on input. Then you use the deferred value hook to

create a deferred version of the state value and use the memo hook to regenerate the `BooksList` component only when the `deferredSearchString` value—that is, the return value of the deferred value hook—changes. Through the console outputs, you can see that the `App` component is rendered more often than the `BooksList` component. You save resources on high-frequency changes that involve expensive operations such as server communications. The `BooksList` component receives the value as a prop and is thus only indirectly affected by the deferred value hook, as you can see in [Listing 6.22](#):

```
import { useEffect, useState } from 'react';

function BooksList({ searchString }) {
  const [books, setBooks] = useState([]);

  useEffect(() => {
    fetch(`http://localhost:3001/books?title_like=${searchString}`)
      .then((response) => response.json())
      .then((data) => setBooks(data));
  }, [searchString]);

  return (
    <ul>
      {books.map((book) => (
        <li key={book.id}>{book.title}</li>
      ))}
    </ul>
  );
}

export default BooksList;
```

Listing 6.22 Display of the Books List (`src/BooksList.jsx`)

6.10 “useTransition”: Lowering the Priority of Operations

Quick Start

The `useTransition` function returns an array with a Boolean value and a function. This `startTransition` function in turn accepts a callback function that marks React as a transition and treats it with low priority. The first element of the return array of the `useTransition` function signals whether the transition is already complete.

```
const [isPending, startTransition] = useTransition();
```

Listing 6.23 Syntax of the “useTransition” Function

The `useTransition` function allows you to assign a lower priority to operations. Here you should especially distinguish between operations that generate direct feedback for the user (e.g., updating a text field when typing) and operations that cause the interface to be updated in some other way and for which you can also briefly display a loading indicator.

Direct feedback to users is important and should always be the highest priority. Delayed text input quickly causes frustration. As a concrete example, you can see a list in [Listing 6.24](#) that you can filter via a text field. Updating the text field has a high priority, whereas updating the list has a lower priority.

```
import { useState, useTransition } from 'react';
import './App.css';

function App() {
  const [searchString, setSearchString] = useState('');
  const [books, setBooks] = useState([]);
  const [isPending, startTransition] = useTransition();

  function search() {
    startTransition(() => {
      fetch(`http://localhost:3001/books?title_like=${searchString}`)
        .then((response) => response.json())
        .then((data) => setBooks(data));
    });
  }

  return (
    <div>
      <div>
        <input
          type="text"
          value={searchString}
          onChange={(event) => setSearchString(event.target.value)}
        />
        <button onClick={() => search()}>filter</button>
      </div>
      {isPending && <div>loading</div>}
      {!isPending && (
        <ul>
          {books.map((book) => (
            <li key={book.id}>{book.title}</li>
          ))}
        </ul>
      )}
    </div>
  )}
```

```
    </div>
  );
}
export default App;
```

Listing 6.24 Using the “useTransition” Function (src/App.jsx)

The `useTransition` function returns an array with two elements. The first element indicates whether the transition is currently in process. You can use this information to either inform users that the list is being loaded or display the list itself when the transition is complete.

You can use the second element of the array returned by the `useTransition` function—the `startTransition` function—to encapsulate the state change in it. React then ensures that the list update gets a correspondingly lower priority when rendering the component.

6.11 “useId”: Creating Unique Identifiers

Quick Start

The `Id` hook generates unique IDs that you can use, for example, to assign labels. The function generates the same IDs during the server-side rendering and in the hydration process.

```
const id = useId()
```

Listing 6.25 Syntax of the “useId” Function

The `useId` function is primarily intended for applications that are rendered on the server side. This is where problems can arise if you generate unique IDs on the server side and the *hydration process*, which takes control of React on the client side, generates different IDs.

To get around this problem, the React development team created the `Id` hook.

However, you can also use this hook in a purely client-side React application. [Listing 6.26](#) shows an example of this.

```
import { useId } from 'react';
import './App.css';

function App() {
  const id = useId();
  return (
    <div>
```

```

    <label htmlFor={id}>Name:</label>
    <input type="text" id={id} />
  </div>
);
}
export default App;

```

Listing 6.26 Using the “useId” Function (src/App.jsx)

The component in the example uses the Id hook to create a unique ID and uses this ID to link the `label` element with the `htmlFor` property to the `input` element.

6.12 Library Hooks

The two library hooks React introduces with version 18 will probably only be used in rare, exceptional cases directly in your application. For this reason, only a brief look at what these hooks can do follows here, as they are aimed more at authors of libraries for React.

6.12.1 “useSyncExternalStore”

The sync external store hook is used when it comes to connecting your application to an external data source. This hook is intended to make the connection to the data source compatible with the new render process. A typical external data source, also called a *store*, can be a Redux store, for example.

When called, the hook function returns a state object that you can use in your components. When you execute the function, you pass a `subscribe` function that you can use to register a callback function for the changes to the store. The second argument is a function that returns the current value of the data source. As an optional third argument, you can pass a function that returns the current value of the data source during server-side rendering of your application.

For example, the *Redux* central state management library uses this hook function for the integration into a React application, replacing the previously used `useMutableSource` function.

6.12.2 “useInsertionEffect”

The insertion effect hook is aimed at authors of CSS-in-JS libraries. Similar to `useEffect` and `useLayoutEffect`, React executes it when changes are made to a component, but before the adjustments are made to the DOM. This allows you to apply styles before the browser renders the elements.

CSS-in-JS

The term *CSS-in-JS* refers to libraries that allow you to write CSS code in JavaScript and benefit from that. Thus, the resulting styles are limited in scope, and you can use program logic to define your styles.

6.13 Custom Hooks

Custom hooks are JavaScript functions that can use hook functions and are themselves used in a React component. The advantage here is that you can use these custom hooks in multiple places in your application. In addition, custom hooks are the only way to use hooks besides function components.

If you implement a custom hook in your application, you should save it in a separate file and export it. In this way, you ensure the best possible reusability. You should also follow the naming convention for hooks and prefix the hook name with the word *use*. To illustrate this, we'll first implement a simple example of a custom hook, detached from the sample application.

In the following example, you implement a counter that displays a value incremented once per second. In this implementation, you separate the logic and the presentation by using a custom hook. Try to make a component only care about the presentation if possible and have little logic of its own. You can solve this by swapping out all the logic into a custom hook. For this purpose, you create a file named *useCounter.js*. The source code of this file is shown in [Listing 6.27](#):

```
import { useState, useEffect } from 'react';

function useCounter() {
  const [counter, setCounter] = useState(0);

  useEffect(() => {
    const interval = setInterval(
      () => setCounter((prevCounter) => prevCounter + 1),
      1000,
      []
    );
    return () => clearInterval(interval);
  }, []);

  return counter;
}

export default useCounter;
```

Listing 6.27 Custom Hook with Counter Functionality (src/useCounter.js)

The counter hook consists of a state hook and an effect hook. You encapsulate both function calls in the `useCounter` function. A special feature of custom hooks is that when you create the hook, you need to consider what the user needs access to. In this case, only read access to the state—that is, the `counter` variable—is required.

You only need the `setCounter` function for the internal logic of the hook. It therefore remains hidden from the outside world. For more complex hooks, you may need to export more than just the state and the function to set the state. Here you can either use an array, as with the state hook, or an object.

The integration of the custom hook into the `App` component is similar to the integration of a base hook: you import the function and execute it within the function component. [Listing 6.28](#) contains the respective source code:

```
import useCounter from './useCounter';

function App() {
  const counter = useCounter();

  return <div>{counter}</div>;
}

export default App;
```

Listing 6.28 Integration of the Custom Hook (`src/App.ts`)

As you can see in the code example, the component becomes much simpler than if you include the two hooks directly there. The effect of the custom hook is similar to that of the individual base hooks. This means that for each call of the `useCounter` function, the state hook creates a standalone state that is independent of the other instances. The same applies to the effect hook: it's triggered whenever the integrating component changes, unless you restrict this with the second parameter. If you return a callback function, it will be executed when you unmount.

In some places in this chapter, we already mentioned that hooks may only be used in function components or custom hooks. The developers of React have dedicated a separate section of the documentation to the rules for hooks, which they called *Rules of Hooks*, which you can find at <https://react.dev/warnings/invalid-hook-call-warning>.

6.14 Rules of Hooks: Things to Consider

Okay, here's a newly developed interface, and it's already regulated. But that's not as bad as it sounds. Overall, there are two rules you should heed when using hooks. To better monitor compliance with these rules, an ESLint plugin is available for you to integrate into your application. If you violate any of the rules, you'll receive a warning on

the console. The plugin is named *eslint-plugin-react-hooks* and is already included in the default configuration of Create React App, so you don't need to install and include it separately.

6.14.1 Rule #1: Execute Hooks Only at the Top Level

You should only execute a hook function like `useState` or `useEffect` directly in the component or a custom hook and not in a nested function, loop, or condition. This ensures that a hook is not executed multiple times and that the hooks are always activated in the same order. Take a component such as the one in [Listing 6.29](#), where the state hook is called within an `if` statement:

```
import { useState } from 'react';

function App() {
  if (true) {
    const [state, setState] = useState(0);
  }
  return <div>Hello World</div>;
}

export default App;
```

Listing 6.29 Component with a State Hook in a Condition

Based on the ESLint rule for the Hooks API, you receive the following error message:

```
React Hook "useState" is called conditionally. React Hooks must be called in the
exact same order in every component render. Eslint(react-hooks/rules-of-hooks)
```

Listing 6.30 Error Message that Appears when a Hook Is Not Used at the Top Level

6.14.2 Rule #2: Hooks May Only Be Used in Function Components or Custom Hooks

The Hooks API is strongly coupled to the function components of React and for this reason should only be used either directly in function components or in custom hooks. In the source code shown in [Listing 6.31](#), the call of the `useEffect` function is swapped out to a separate function. Such a move violates the second rule.

```
import { useEffect } from 'react';

function init() {
  useEffect(() => {
    console.log('Effect hook');
  }, []);
}
```

```
function App() {
  init();
  return <div>Hello World</div>;
}

export default App;
```

Listing 6.31 Component with an Effect Hook in an External Function

A run of ESLint on this component returns the following error message:

```
React Hook "useEffect" is called in function "init" that is neither a React func
tion component nor a custom React Hook function. React component names must star
t with an uppercase letter. React Hook names must start with the word "use". esl
int(react-hooks/rules-of-hooks)
```

Listing 6.32 Error Message that Appears when a Hook Is Called in a Function

In this example, the problem would be easily solved by renaming the function. In this case, only the `use` prefix is missing to convert the function to a custom hook. However, if the `init` function is a classic helper function that contains other functionality besides the effect hook, the violation becomes even clearer.

6.15 Changing over to Hooks

The Hooks API has become an integral part of the feature set of React. This doesn't mean that class components will disappear from the library in the foreseeable future. However, the combination of hooks and function components has supplanted class components as the standard for how components are built in React. One of the ulterior motives for introducing the Hooks API was to simplify concepts and reduce the barrier to entry.

The developers of React recommend that existing applications—especially if they have a larger range of functions—do not immediately and completely switch to hooks. Instead, such a changeover, if it makes sense, should only be made in small steps. New developments of extensions and modules, in which hooks are introduced, are the first choice for this. Then the components of the application can be changed step by step. A viable solution to this is to have a component be converted as soon as its source code is changed in the course of developing a new feature or a bug fix.

Many of the existing concepts of React are picked up by the Hooks API and thus remain almost unchanged. A good example is the context hook, which gives you convenient access to the context, similar to a class component. The Context API remains unchanged, but the functional components are significantly upgraded at the same time.

6.16 Summary

In this chapter, you learned how the Hooks API of React significantly impacts the way you use components:

- The Hooks API enables you to reduce the size of your components, eliminate duplicates in your code, and implement complex design patterns much more easily.
- Due to the Hooks API, class and function components are equivalent.
- The Hooks API is divided into the following three parts: base hooks, advanced hooks, and library hooks.
- The base hooks are the state hook, the effect hook, and the context hook. You will use these most of the time.
- The extended hooks add additional functionality, such as the ability to memoize structures or intervene in the render process of React.
- You generally use the library hooks only when you implement libraries for React.
- You can group the hooks into custom hooks to swap them out from your components and make them reusable.
- Hooks may only be used at the top level and only in components or custom hooks.

In the next chapter, you'll learn how to bring more structure to your React application using TypeScript.

Chapter 7

Type Safety in React Applications with TypeScript

JavaScript has a weak type system, which supports only a handful of types. It isn't possible to assign a fixed type to variables, just as you cannot assign types to the signature of functions. The more extensive an application becomes and the greater the number of developers who are involved in the implementation of the application, the more the lack of a strict type system in JavaScript becomes noticeable.

To address this problem, there are type systems based on JavaScript. There are two different categories here: static type checking and languages compiled to JavaScript. In this chapter, we'll introduce Flow and TypeScript, representatives of each of these two approaches, and demonstrate how they work with React.

7.1 What Is the Benefit of a Type System?

Before you integrate one of the type systems available on the market into your application, you should first ask yourself: How will I benefit from a type system? In general, you get additional structure and safety. By specifying what type a particular variable, parameter, or return value of a function has, you do limit the freedom that JavaScript gives you. But it also means that you can no longer accidentally turn a string variable into a number or a Boolean.

If you strictly follow the type system and specify the types for every variable and function, it also forces you to think more about the structure of your application. At the same time, this requires you to document your code. A comment block of a function can easily become obsolete because you are not forced to adjust it when the source code changes. When using a type system, you must adjust the type specification in the signature of a function; otherwise you will receive error messages during checking.

The readability of the source code is positively influenced by the use of a type system. When looking at the signature of a function, you can see at a glance what it expects as input and what it returns. If you then combine the types with a meaningful naming of the function itself as well as the parameters, even someone who did not write the function himself should be able to recognize at a glance what it does.

A type system can also serve well in regard to troubleshooting and in the maintenance of applications as it reduces the risk of problems arising with complex dependencies because they are stored directly in the source code.

The most obvious advantage of a type system is improved support via programming tools such as the development environment or tools for static code analysis. Most development environments support the common type systems by default or have extensions that can be installed in a few steps. If your development environment is configured correctly, you will receive immediate feedback on your source code during development. This is done on the one hand by error messages if you violate the rules of the type system, and on the other hand by autocompletion during development—for example, when you start writing the name of a method of an object and get possible suggestions. This feature exists for native JavaScript as well, but it's significantly better and more reliable with the type-safe variant.

7.2 The Different Type Systems

As mentioned previously, type systems for JavaScript can be roughly divided into two categories: tools that use certain annotations to check compliance with the rules, and full-fledged type systems, where the source code is formulated in its own language. In the first category, the source code is already in JavaScript and only the type information is added. Before the source code can be executed, the type specifications must be removed; otherwise syntax errors would be thrown. A typical representative of this type is Flow.

The second type of type system, which includes TypeScript, for example, also uses annotations to check compliance with type rules. The TypeScript code is compiled into JavaScript before execution. This removes the type specifications and emulates certain features. The TypeScript compiler is capable of generating different versions of JavaScript—both modern code, as it can only run in a modern browser, and older JavaScript source code that follows the specification of ECMAScript 3 or ECMAScript 5.

7.3 Type Safety in a React Application with Flow

Flow was released by Facebook in 2014 and has since been managed as an open-source project on GitHub. The website for Flow can be found at <https://flow.org/>. Flow was developed to improve the quality of source code. In addition, the productivity of developers should be significantly increased by the static types in JavaScript.

Flow was designed to be a lightweight tool for development. The basic character of JavaScript should remain as unchanged as possible and only be supplemented by the support of types. An important requirement for type checking with Flow is that this process is done very quickly so that it doesn't have a negative impact on the

development process. Flow checks the types based on files. This allows the verification to be performed simultaneously in multiple files. It's also possible to mark only certain files for checking.

Like most type systems, Flow provides a feature called *type inference*. Here, Flow derives type information from existing source code that isn't tagged with types. This means that you don't necessarily have to mark up all source code with types: once the type is apparent from the source code, you can omit the explicit assignment.

Flow is being developed in parallel with JavaScript so that modern interfaces and features are also supported.

7.3.1 Integration into a React Application

Both React and Flow come from Facebook, so it's not surprising that Flow can be integrated into a React app created with Create React App in just a few steps.

For the short introduction to Flow that follows, you first want to create a new application with Create React App using the `npx create-react-app flow-test` command. Then you need to install the `flow` package using `npm install --save-dev flow-bin` and configure it. For this purpose, you must first add a new entry to the `scripts` section with the `flow` key and `flow` value in your `package.json` file. This way you make sure that you can run Flow on the command line using the `npm run flow` command.

The `npm run flow init` command enables you to generate the configuration for Flow, which the tool stores in the `.flowconfig` file in the root directory of the application.

However, before you can run your application in the browser, you must remove the Flow-specific extensions to the source code because they are JavaScript syntax violations. You can either let Babel do this cleanup or you can use the `flow-remove-types` tool, which you can install via the `npm install --save-dev flow-remove-types` command. You can integrate the processing of your source code into the NPM scripts of your application, as shown in [Listing 7.1](#):

```
{
  ...
  "scripts": {
    "flow": "flow",
    "prestart": "flow-remove-types src/",
    ...
  },
  ...
}
```

Listing 7.1 Extension of the “package.json” File for Flow

Create React App supports the new versions of Flow directly, which means you don't need to install `flow-remove-types` separately.

After installing Flow, you can use the tool in your application. [Listing 7.2](#) contains an App component with Flow support enabled:

```
// @flow
import React from 'react';
import type { Node } from 'react';

function App(): Node {
  let name: string = 'World';

  name = 1337;
  return (
    <div className="App">
      <h1>Hello {name}</h1>
    </div>
  );
}

export default App;
```

Listing 7.2 Component with Active Flow Support (src/App.jsx)

In the first line, you use the `@flow` comment to enable checking via Flow. Without this comment, Flow would ignore this file. The second part in the source code relevant for Flow is the declaration of the `name` variable. Here you can see that the variable `name`, separated by a colon, is followed by the specification of the type. The subsequent assignment of the number 42 is an obvious violation of the type system and generates a corresponding error when Flow is executed.

Despite the error and the syntax violation due to the type specification, the application remains executable. You can easily test this by entering the `npm start` command on the command line in the root directory of your application.

You can run the check directly in your development environment, on the command line, or with a combination of both approaches.

Execution in the Development Environment

Both WebStorm and Visual Studio Code provide support for Flow. In the case of WebStorm, this is already included but still needs to be configured by changing the language version of your application to Flow. The only requirement you then need to fulfill is to make sure that Flow is installed in your application.

Using Flow in Visual Studio Code is similar. Here you install the Flow extension, which is named *Flow Language Support*. For this to work correctly, you must set the `javascript.validate.enable` key to the value `false` in the IDE settings; otherwise the flow statements will be considered syntax errors. Again, the flow package must be

installed locally in your application. [Figure 7.1](#) contains the source code from [Listing 7.2](#) in Visual Studio Code. Here the number 1337 is marked in red to indicate an error. If you move the mouse over the number, you'll see the corresponding error message, which will provide you with additional information about the error and a possible fix.

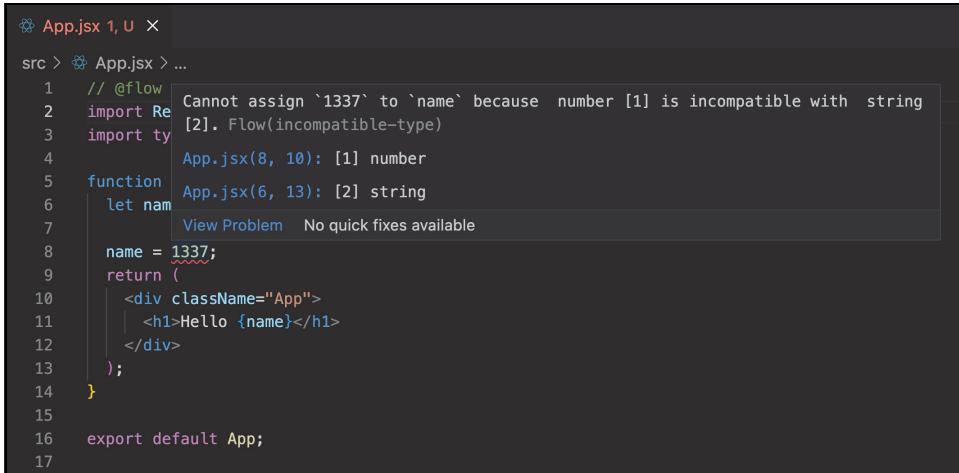


Figure 7.1 Error Message in Visual Studio Code

Execution on the Command Line

The integration of Flow with the development environment ensures that you get instant feedback as you generate source code. Errors are displayed immediately and can be corrected directly. However, for use in a build or release process, you should resort to the command line tool. For this purpose, you want to go to the root directory of your application in the command line and execute the `npm run flow` command there. As a result, you'll get output like that shown in [Listing 7.3](#):

```
$ npm run flow
```

```
> flow-test@0.1.0 flow
> flow
```

```
Error ----- src/App.jsx:8:10
```

```
Cannot assign 1337 to name because number [1] is incompatible with string
[2]. [incompatible-type]
```

```

5| function App(): Node {
[2] 6|   let name: string = 'World';
7|
[1] 8|   name = 1337;
9|   return (

```

```
10|    <div className="App">
11|        <h1>Hello {name}</h1>
```

Found 1 error

Listing 7.3 Execution of Flow in the Command Line

An exit code other than 0 indicates a failure of the command. Thus, the tool can be integrated into an automated build process. The build process can be aborted if the source code has type system violations.

7.3.2 The Major Features of Flow

Besides the base types, such as strings, numbers, or Booleans, arrays and objects are also supported. Also, in Flow, each class you create is a separate type, which you can specify. In addition, you can use union types to create a combination of multiple types using the pipe operator.

Flow also supports generics and interfaces. *Generics* allow you to create generic data structures and make them concrete only when you use them. A typical example of generics is a *collection*—that is, a collection of objects of a certain type. *Interfaces* define structures without concrete implementations. You can use them to indicate that an object must have a certain structure.

7.3.3 Flow in React Components

Flow doesn't show its advantages in combination with React only when typifying variables and function signatures, but also when defining components. The critical parts are the definition of props and states.

For the props, you already learned about a possible solution: `PropTypes`. However, there is no solution for defining the structure of the state of a component. For a function component, you can set the props as the type in the parameters list and the return value as the regular return value of the function. There are also type definitions for all type-dependent hooks, such as the state hook. In [Listing 7.4](#), you can see a `BooksList` component as an example. This component gets its initial value via the `initialBooks` prop and then stores it in the state. The `useState` function is typed as a generic function, so you can specify the type explicitly. In most cases this isn't necessary because Flow can derive the type from the initial value. But if you have the option, be as explicit as possible about type specifications.

```
// @flow
```

```
import React, { useState } from 'react';
import type { Node } from 'react';
```

```

type Book = {
  id: number,
  title: string,
  author: string,
  isbn: string,
  rating: number,
};

type Props = {
  initialBooks: Book[],
};

function BooksList({ initialBooks }: Props): Node {
  const [books, setBooks] = useState<Book[]>(initialBooks);

  return (
    <ul>
      {books.map((book) => (
        <li key={book.id}>{book.title}</li>
      ))}
    </ul>
  );
}

export default BooksList;

```

Listing 7.4 “Counter” Component with Flow (src/BooksList.jsx)

To enable Flow, you add the `@flow` comment at the beginning of the file. Then you define a type alias for each of the book records and the component's props. The state contains the initial book records. When you call the `useState` function, you specify the `Book[]` type so that Flow can ensure for itself that you can only work with a list of books. If you violate the type specifications during implementation, you'll receive corresponding error messages. You can run the application as usual via `npm start`, despite the type specifications. The Flow-specific parts are removed by the build process.

In addition to Flow, TypeScript is another widely used type system that is also very popular in the React community.

7.4 Using TypeScript in a React Application

TypeScript has been developed by Microsoft as an open-source project since 2012. The way it works is fundamentally different from Flow. Whereas Flow focuses on type

markup and leaves the rest of the source code untouched, TypeScript is a programming language in its own right that extends the core of JavaScript and adds other features.

Basically, valid JavaScript code is also valid TypeScript code. However, this statement isn't true in the other direction. Trying to run TypeScript directly in the browser usually results in syntax errors and termination of the application. TypeScript has become the de facto standard for type-safe JavaScript in recent years.

In any case, you should consider using a type system for your application, as the advantages clearly outweigh the disadvantages. As a type system, we clearly recommend TypeScript, as it has a much higher penetration in the community and is also very well supported by the various add-on libraries for React.

7.4.1 Integrating TypeScript in a React Application

The combination of React and TypeScript has been supported for quite some time. TypeScript wasn't officially included in Create React App until version 2.1 in October 2018. Since the release of this version, it's possible to start the development of an application directly with TypeScript. Meanwhile, the Create React App team has tweaked the architecture a bit so that you can specify a template for creating your app. One of the default templates is named `typescript` and initializes a React application with TypeScript support for you. [Listing 7.5](#) shows the command you use to initialize the application with TypeScript:

```
npx create-react-app library --template typescript
```

Listing 7.5 Initializing an Application with TypeScript

By using the `--template typescript` option, you make sure that Create React App installs all the dependencies required to use TypeScript in your application. It also creates the necessary structures and configurations so that you can start developing right away. The development and build process is also modified accordingly.

The first difference from an ordinary React application is the presence of the `tsconfig.json` file in the root directory, which can be used to configure the behavior of TypeScript. Also, unlike Flow, TypeScript doesn't give you the option to selectively check only part of the files; you have to do this with all the files in your application. The component files also have the *extension* `.tsx`, which should indicate that it is a combination of TypeScript and JSX. For helper files that don't contain JSX, you can use the `.ts` extension. For demonstration purposes, you can use the source code from [Listing 7.2](#):

```
import React from 'react';
import './App.css';

const App: React.FC = () => {
  let name: string = 'World';
```

```
name = 42;  
return (  
  <div className="App">  
    <h1>Hello {name}</h1>  
  </div>  
);  
}
```

```
export default App;
```

Listing 7.6 TypeScript Source Code (src/App.tsx)

When you use TypeScript, you can omit the explicit specification of the type. The first assignment of a value to a variable simultaneously determines its type. Otherwise, TypeScript behaves very similarly to Flow in the case of faulty source code, as in this example.

Execution in the Development Environment

TypeScript is supported by all major development environments, such as WebStorm or Visual Studio Code. The source code is checked immediately upon its creation, and errors are displayed right away. With this direct feedback, many errors can be found and fixed before the source code is executed. Also, the suggestions that the development environment gives you when writing the source code are significantly better than when using pure JavaScript because the signature of functions and the structure of objects are known. [Figure 7.2](#) shows the error message you get for the source code from [Listing 7.6](#) when you open it in Visual Studio Code.

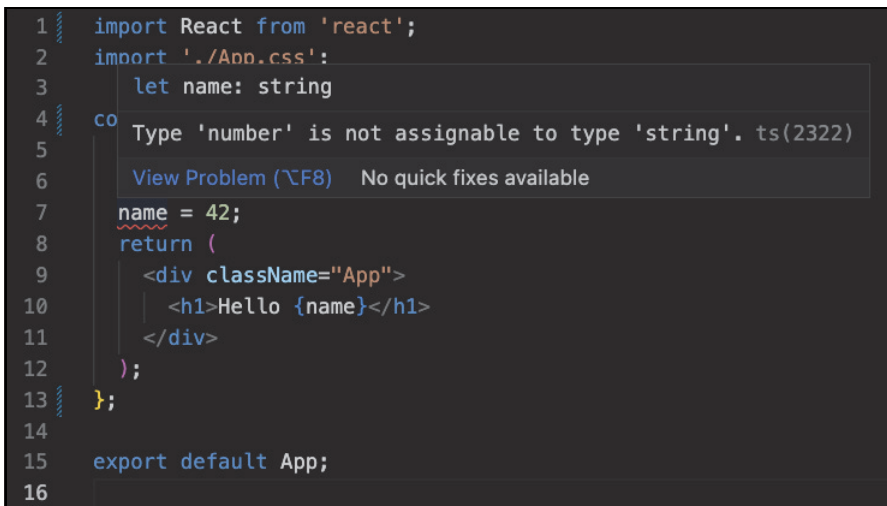


Figure 7.2 TypeScript Error Message in Visual Studio Code

You can use TypeScript not only in the development environment, but also in the command line, and thus integrate it into an automated build process, for example.

Execution in the Command Line

At the core of TypeScript, there is the *TypeScript compiler* or `tsc`. This command-line tool checks the TypeScript source code of your application and transforms it into valid JavaScript source code that can be run in the browser. For this transformation to work, the `typescript` package must be installed on your system. Create React App will do the installation for you. After the installation, you can write TypeScript and transform the code using the default TypeScript configuration. Alternatively, you can use the `tsc --init` command to have a configuration generated in the form of `tsconfig.json`. This affects the way the TypeScript compiler works. In the current example, such a configuration file already exists, and you only need to switch to the command line and enter the `npx tsc` command. The compiler automatically finds the configuration file and applies it. In [Listing 7.7](#), you can see the output in the command line.

```
$ npx tsc
src/App.tsx:7:3 -
  error TS2322: Type 'number' is not assignable to type 'string'.
```

```
7   name = 42;
   ~~~~
```

```
Found 1 error in src/App.tsx:7
```

Listing 7.7 Output of the TypeScript compiler in the Command Line

The current configuration of TypeScript ensures that no JavaScript source code is stored in the file system during the development process. This isn't necessary because TypeScript is very deeply integrated into the development process. With `npm start`, you run the application directly based on Webpack with the dev server. For production use, you need to have the source code translated to JavaScript. The `npm run build` command will help you with this. It creates an executable application for you, which you can deploy to any web server. With this knowledge, you can now dive deeper into the world of TypeScript.

The following explanations and examples of TypeScript are significantly more extensive than those for Flow as the rest of the book is based on TypeScript. The reasons for this are mainly that TypeScript is much more powerful compared to Flow and that it has a wider distribution.

7.4.2 Configuration of TypeScript

You can influence the behavior of the TypeScript compiler through options in the command line or through the *tsconfig.json* file. If such a file exists in the root directory of your application, where you also run the compiler, the file is used automatically and you don't need to reference it explicitly.

The most important specifications in the configuration are the `compilerOptions`, which you can use to control the compiler, and `include`, which you can use to specify a list of directories that contain the files with the TypeScript source code. If necessary, you can use the `exclude` key to exclude certain patterns from the compilation process and use `files` to specify individual files. In [Listing 7.8](#), you can see the default configuration for TypeScript that Create React App generates for you:

```
{
  "compilerOptions": {
    "target": "es5",
    "lib": [
      "dom",
      "dom.iterable",
      "esnext"
    ],
    "allowJs": true,
    "skipLibCheck": true,
    "esModuleInterop": true,
    "allowSyntheticDefaultImports": true,
    "strict": true,
    "forceConsistentCasingInFileNames": true,
    "noFallthroughCasesInSwitch": true,
    "module": "esnext",
    "moduleResolution": "node",
    "resolveJsonModule": true,
    "isolatedModules": true,
    "noEmit": true,
    "jsx": "react-jsx"
  },
  "include": [
    "src"
  ]
}
```

Listing 7.8 TypeScript Configuration of a React Application (tsconfig.json)

Usually, the default configuration should be sufficient for using React, so you'll hardly come into contact with the configuration during the development process.

7.4.3 The Major Features of TypeScript

The differences in source code markup and key features are minor between Flow and TypeScript. However, the two differ seriously in the basic handling of the source code. TypeScript also supports basic data types such as `Boolean`, `string`, or `number` as well as composite data types such as objects and arrays. In addition, there are special data types such as `enum`, which you can use to map keys to values.

TypeScript allows the use of classes that represent types at the same time and thus can be used in type assignments. Type aliases, interfaces, and generics are also supported by TypeScript and used in React applications. In TypeScript, you can use *modules*, which are self-contained units that each reside in a file. The module syntax is the same as in the ECMAScript module system, so you can still use the `import` and `export` keywords in your application to import and export your modules, respectively.

7.4.4 Type Definitions: Information about Third-Party Software

TypeScript has a special feature when dealing with libraries written in JavaScript that are to be used in a TypeScript application. The interfaces of these libraries are usually not provided with type information, so you would lose most of the benefits of TypeScript at this point as the compiler would fall back to the implicit `any` type. If you use `any`, further type checking is no longer possible at this point.

To solve this problem, TypeScript provides a feature called *type definitions* that can be used to provide types to JavaScript interfaces. An increasing number of libraries in the React environment ship these type definitions right along with the actual source code so that the versions of the interface and the type definition are identical. Far more often, however, the type definitions are offered as an additional package. This creates the risk that the versions are different and you are working with a potentially outdated version of the interface definition. But type definitions are often also maintained by the people who created the libraries, so few problems are expected here.

DefinitelyTyped has established itself as the most important source. This is a repository from which you can obtain type definitions for numerous libraries. Type definitions are maintained in a GitHub repository (<https://definitelytyped.github.io/>) and can be selectively installed via a package manager such as Yarn or NPM. When installing a type definition, you want to make sure to install it as `devDependency`. Type definitions are used only during development and not for running your application.

To install the type definition for React, which Create React App automatically does for you, you need to run the `npm install -D @types/react` command. The `@types` prefix stands for the DefinitelyTyped repository. After the installation, you don't need to do anything else: you can work directly with the type definitions because TypeScript automatically looks for an `@types` directory in `node_modules` if there are no direct type definitions for a particular library.

With this knowledge of TypeScript, you can now move on to integrating TypeScript into your application step by step.

7.5 TypeScript and React

To use TypeScript in your application initialized with Create React App, you have two options: you can start directly with TypeScript and specify the `--template typescript` option when initializing your application, or you can add TypeScript to your existing project. In the following sections, we'll first demonstrate how you can use TypeScript by means of an independent example. Then the key aspect of the sample application is converted to TypeScript.

7.5.1 Adding TypeScript to an Existing Application

The sample application from the previous chapters is based on JavaScript. However, you can integrate TypeScript with a few steps. For this, you must first install some additional dependencies. [Listing 7.9](#) contains the appropriate command:

```
npm install typescript \n  @types/node \n  @types/react \n  @types/react-dom \n  @types/jest
```

Listing 7.9 Installing Dependencies for TypeScript

After the installation, you need to rename the JavaScript files in the *src* directory: The *.js* extension becomes *.ts*, and *.jsx* becomes *.tsx*. After that, you must restart the development server. The server recognizes that this is a TypeScript project after the changes and automatically creates a *tsconfig.json* file for you.

But renaming the files alone doesn't turn your JavaScript application into a TypeScript application. This becomes clear when you look at the console where you started the development server. There you'll see errors that cause your application to fail to start. In the following sections, we'll troubleshoot these error messages. During this, you'll learn about some features of TypeScript that are important for developing a React application.

7.5.2 Basic Features

The basics of TypeScript are the same no matter what environment you're in. That's why we'll first look at the basics of TypeScript before you get to know the React-specific parts.

Variables

In TypeScript, you declare a variable as you would in JavaScript, except that you can still specify an optional type. If you initialize the variable with a value, you can omit the type assignment because TypeScript automatically takes the type of the assigned value. The `var`, `let`, and `const` keywords are available for the declaration. If possible, you should only use `let` and `const`. Try to use `const` as often as possible to avoid accidental reassignment. Only when you need to reassign a value or know this in advance should you use the `let` keyword. The `const` and `let` keywords create constants and variables respectively in the block scope, so you have very good control over validity. [Listing 7.10](#) shows an example of a declaration and simultaneous initialization of a variable, where you also specify the type:

```
let title: string = 'Design Patterns';
```

Listing 7.10 Declaration and Initialization of Variables

You always write the specification of the type after a colon separating it from the name of the variable and before the equal sign. [Table 7.1](#) contains an overview of the data types of TypeScript.

Type	Description
boolean	The truth values <code>true</code> and <code>false</code>
number	Numerical values like integers, floats, and <code>BigInt</code>
string	Strings
array	Corresponds to the array type of JavaScript
tuple	A fixed length array with fixed assigned types
enum	An enumeration type where keys are mapped to numbers or optionally other values
unknown	The content of a variable of this type can be of any type
any	Turns off type checking for a variable
void	Absence of a value, such as in the case of a function without a return value
null	The <code>null</code> value of JavaScript
undefined	The <code>undefined</code> value of JavaScript
never	A value that never occurs, such as a function that throws an exception in every case
object	Stands for a JavaScript object

Table 7.1 Basic Data Types in TypeScript

Functions

Functions take a prominent role in the development of React applications. In a React application, you usually deal with functions much more often than with class constructs. Functions can appear as an arrow function, a named function, or an anonymous function. For TypeScript, the signature of the function is relevant—that is, the parameters list and the return value.

```
function add(a: number, b: number): number {
  return a + b;
}

const getFullname = function (firstname: string, lastname: string): string {
  return `${firstname} ${lastname}`;
};

const greet = (name: string): string => {
  return `Hello ${name}`;
};
```

Listing 7.11 Examples of Functions in TypeScript

Listing 7.11 contains examples of the following:

- A *named function*—that is, a function that has its own name
- An *anonymous function*—that is, a function that has no name and that you assign to a constant
- An *arrow function*

No matter what type of function you decide to use, try to be as explicit as possible and always specify the types for the parameters and the return value. Especially the return value can save you from careless mistakes—for example, if you forget a `return` statement.

Classes

Even though functions are clearly gaining importance over class constructs in the React world, there are always places where it pays off for you to rely on classes instead. Classes are used especially for data encapsulation, the implementation of certain business logic, and, last but not least, for class components.

A TypeScript class behaves very much like a class in modern JavaScript. To define it, you use the `class` keyword followed by the name of the class, which should start with an uppercase letter according to the naming convention. After the class name, you can use the `extends` keyword and specify a class name to inherit from that class, or you can use the `implements` keyword and specify an interface that the class must then implement. The use of interfaces is the first major difference from traditional JavaScript.

In a TypeScript class, you can define a *constructor*. This is a special method that's called by TypeScript when you create a new instance of the class via `new`. The name of the constructor is `constructor`. In the constructor, you can define a parameters list. These values are passed during the instantiation.

Besides the constructor, you can define properties and methods in a TypeScript class. Methods follow the same rules for specifying the signature as functions do. For properties and methods, you can specify their visibility using the access modifiers: `private`, `public`, and `protected`.

Access Modifiers in TypeScript

Unlike native JavaScript, TypeScript supports access modifiers that let you affect the visibility of the properties and methods of a class. TypeScript has the three modifiers—`private`, `protected`, and `public`—which are also known in other programming languages:

- **private**

Properties and methods marked with the `private` modifier can only be accessed within the class. This means that they aren't available outside, nor in derived classes.

- **protected**

A property marked as `protected` can be used in the class and its subclasses.

- **public**

The default modifier in TypeScript is `public`. If you don't specify a modifier, the property or method is automatically `public` and can be used anywhere in your application.

Another modifier you can use in a class definition is `readonly`. Properties marked with `readonly` must be initialized in the constructor or when they are declared and cannot be changed later.

If you define a parameters list in the constructor and you want to assign these values to specific properties of the class, you should perform this operation directly in the constructor. TypeScript provides a shortcut for this very common use case: if you specify a combination of access modifier, property name, and type for a parameter, TypeScript automatically assigns that value to the specified class property and you don't have to worry about anything else. This shorthand notation is called *parameter properties*.

In [Listing 7.12](#), you can see an example of a TypeScript class including instantiation and method call:

```
class User {  
  constructor(private firstname: string, private lastname: string) {}  
  
  get fullname(): string {
```

```

    return `${this.firstname} ${this.lastname}`;
  }

  greet(greeting: string): string {
    return `${greeting} ${this.fullname}`;
  }
}

const tom = new User('Tom', 'Miller');
const greeting = tom.greet('Hello');
console.log(greeting); // Hello Tom Miller

```

Listing 7.12 Class in TypeScript

Type Aliases versus Interfaces

In TypeScript, you have several options to define your own types. You’ve already learned about classes, which represent one of these options. Two other variants are *type aliases* and *interfaces*. You can use both of them to specify types, such as in variable declarations or in function signatures, but they differ on some points:

- You can extend interfaces by means of *interface merging*. If you define an interface multiple times, TypeScript merges these definitions into one interface.
- Interfaces can inherit from other interfaces via the `extends` keyword. For types, you can use the `&` operator to use a type as a base and add more information to create a new type.
- Type aliases can’t be changed once they’ve been defined.

For simple cases and when you don’t need to ensure that a class implements an interface, a type alias is sufficient in most cases.

With this basic knowledge of TypeScript, let’s now turn to something more React-specific.

7.5.3 Function Components

Quick Start

In a *function component*, you use the generic `React.FC` type and pass the props structure to it. This usually looks like the example in [Listing 7.13](#):

```

import React from 'react';

type Props = {
  title: string;

```

```
};

const Headline: React.FC<Props> = ({ title }) => {
  return <h1>{title}</h1>
}

export default Headline;
```

Listing 7.13 Typing in a Function Component

For function components, TypeScript mainly affects the signature of the function. React, or rather the type definition of React, provides the generic `React.FC` type for function components. This type has come under criticism for some time because it always provides child components for a component even though the component doesn't use them, which has been somewhat misleading in places. But this problem has been solved by this point, and so nothing stands in the way of using this type.

For a function component, you should define a type for the props and store it as a separate type alias within the file. Unless you use the props elsewhere in your application, you shouldn't export them unnecessarily. As a type for the function component itself, you can use `React.FC`, which is implemented as a generic type and accepts the structure of the props. `React.FC` is responsible for defining the correct return type for the function component. As an example of a typical function component, see the `BooksListItem` component in [Listing 7.14](#), which is responsible for displaying a data record in a list. This component receives the data record as a prop and represents the title of the record in a `li` element.

```
import React from 'react';
import Book from './Book';

type Props = {
  book: Book;
};

const BooksListItem: React.FC<Props> = ({ book }) => {
  return <li>{book.title}</li>;
}

export default BooksListItem;
```

Listing 7.14 Typed Function Component (`src/BooksListItem.tsx`)

In this component, you reference the `Book` type in the props. You usually need such types more than once in an application, so you should store it in a separate file, in this

case in the *src* directory and named *Book.ts*. You can see the source code of this file in [Listing 7.15](#):

```
export default type Book = {
  id: number;
  title: string;
  author: string;
  isbn: string;
  rating: number;
};
```

Listing 7.15 Book Type (*src/Book.ts*)

The *BooksItemList* component must receive the record it's supposed to display from its parent component. The parent component in turn must then also take care of the state of the list.

The State Hook

Quick Start

The *useState* component is defined as a generic component. This means you can specify the type of the state separately. Alternatively, you can use the *type inference* of TypeScript, which is TypeScript's ability to infer a type from given type specifications. In this case, TypeScript derives the type from the initial value of the state:

```
const [state, setState] = useState<string[]>([]);
```

Listing 7.16 Syntax of the “*useState*” Function in TypeScript

The state hook is the first base hook where TypeScript becomes relevant as you don't need to specify types in the effect hook. The type definition of the *useState* function provides that it's a generic function. This means that you can specify the type the component works with in the angle brackets after the function name. The *BooksList* component from [Listing 7.17](#) reads the data within an effect hook from the server, writes it to the state, and takes care of rendering the child components:

```
import React, { useState, useEffect } from 'react';
import { Book } from './Book';
import BooksListItem from './BooksListItem';
import axios from 'axios';

const BooksList: React.FC = () => {
  const [books, setBooks] = useState<Book[]>([]);

  useEffect(() => {
```



```
    axios
      .get<Book[]>('http://localhost:3001/books')
      .then((response) => setBooks(response.data));
  }, []);

  return (
    <ul>
      {books.map((book) => (
        <BooksListItem key={book.id} book={book} />
      ))}
    </ul>
  );
}

export default BooksList;
```

Listing 7.17 Managing the Local State of a Component with TypeScript (src/BooksList.tsx)

For the example to work, you need a *data.json* file in the root directory of your application that contains the data. Then you can start the process for the server interface via `npx json-server -p 3001 -w data.json` in the command line. In addition, you still need to install the Axios library using the `npm install axios` command.

When calling the `useState` function, you have to specify the type in this case as the initial value is an empty array. In TypeScript, you can define an array of `Book` objects, either using the more common variant as `Book[]`, or you can use the generic array notation `Array<Book>`. Using the Axios library has the advantage that you can specify in the request what type you expect as a response from the server. This is only a support for typing in your application and doesn't help you validate the data structure that the server sends at runtime.

The final step involves including the `BooksList` component in your `App` component so that React renders the list correctly.

7.5.4 Context

Quick Start

The `createContext` function is defined as a generic function to which you can pass the type of context when you call it:

```
const Context = React.createContext<string[]>([]);
```

Listing 7.18 Syntax of the “createContext” Function

The third basic hook in addition to `useState` and `useEffect`, the context hook, also uses optional type specifications. Note, however, that here you don't specify the type when calling the hook function, but one step prior to that—that is, when creating the context via the `createContext` function. Again, TypeScript is able to infer the type through its type inference feature.

```
import { createContext, Dispatch, SetStateAction } from 'react';
import { Book } from './Book';

const BooksContext = createContext<
  [Book[], Dispatch<SetStateAction<Book[]>>]
>([[], () => {}]);

export default BooksContext;
```

Listing 7.19 React Context with TypeScript (src/BooksContext.tsx)

Typing is a bit more complex in this case, as you store the return value of the state hook in the context to access it from anywhere in your application. The types used are specified by the return value of the `useState` function. As a default value for the context, you define an array with an empty array and an empty arrow function. React will use this structure only if you don't define a provider in the component tree. In the next step, you implement such a provider component that manages the state and assigns it to the provider value:

```
import React, { ReactNode, useState } from 'react';
import { Book } from './Book';
import BooksContext from './BooksContext';

type Props = {
  children: ReactNode;
};

const BooksProvider: React.FC<Props> = ({ children }) => {
  const bookState = useState<Book[]>([]);
  return (
    <BooksContext.Provider value={bookState}>
      {children}
    </BooksContext.Provider>
  );
}

export default BooksProvider;
```

Listing 7.20 Provider with TypeScript (src/BooksProvider.tsx)

The `BooksProvider` component from [Listing 7.20](#) again has a special feature. You use the provider as a node in the component tree to manage the state and pass it a hierarchy of components and elements when you include it in the component tree. You can access this hierarchy via the `children` prop. In TypeScript, you explicitly define the `children` prop in the props type and assign the `ReactNode` type. You then include the props in the `React.FC` type as before. This allows you to specify the child elements in the JSX structure, and React will integrate them correctly in the application.

You should integrate the provider as close as possible to the root component of your application. In the example, this is directly in the `App` component. The corresponding code is shown in [Listing 7.21](#):

```
import React from 'react';
import './App.css';
import BooksList from './BooksList';
import BooksProvider from './BooksProvider';

const App: React.FC = () => {
  return (
    <BooksProvider>
      <BooksList />
    </BooksProvider>
  );
}

export default App;
```

Listing 7.21 Integration of the Context in the “App” Component (src/App.tsx)

In the final step, you customize the `BooksList` component to use the context instead of its local state. In [Listing 7.22](#), you can see how this works:

```
import React, { useEffect, useContext } from 'react';
import { Book } from './Book';
import BooksListItem from './BooksListItem';
import axios from 'axios';
import BooksContext from './BooksContext';

const BooksList: React.FC = () => {
  const [books, setBooks] = useContext(BooksContext);
  useEffect(() => {
    axios
      .get<Book[]>('http://localhost:3001/books')
      .then((response) => setBooks(response.data));
  }, []);
```

```

    return (
      <ul>
        {books.map((book) => (
          <BooksListItem key={book.id} book={book} />
        ))}
      </ul>
    );
  }
}

export default BooksList;

```

Listing 7.22 Integration of the Context in the “BooksList” Component (src/BooksList.tsx)

Because the state and context have the same structure in this case, you don't need to do anything else at this point except for exchanging the `useState` and `useContext` functions.

7.5.5 Class Components

Class components have their own state by default: You can pass props from outside and define lifecycle methods. If you use TypeScript, some things in the implementation will change for you. The basic structure of the class remains unchanged, but you can specify the structure of the state and props and no longer need to use `PropTypes` for this. You can also draw on the full range of TypeScript features when implementing the class, such as access modifiers.

The following example implements a `counter` component as a class component. The component expects you to pass it a `start` value via a `start` prop. Every second, this value is increased by the value 1:

```

import React, { ReactElement } from 'react';

type Props = {
  start: number;
}

type State = {
  counter: number;
}

export default class Counter extends React.Component<Props, State> {
  private interval = 0;

  constructor(props: Props) {
    super(props);
  }
}

```

```
    this.state = {
      counter: props.start,
    };
  }

  componentDidMount(): void {
    this.interval = window.setInterval(() => {
      this.setState((prevState) => ({
        ...prevState,
        counter: prevState.counter + 1,
      }));
    }, 1000);
  }

  componentWillUnmount(): void {
    clearInterval(this.interval);
  }

  render(): ReactElement {
    return <div>{this.state.counter}</div>;
  }
}
```

Listing 7.23 “Counter” Component as a Class Component in TypeScript (src/Counter.tsx)

As you can see in [Listing 7.23](#), you first define the structures for props and state as types. Because you need both structures only in the current file, you don’t need to export either. Then you pass the two types to the generic base class, `React.Component`. This ensures that the structure of the props and the state will be maintained. For example, if you include the `counter` component and don’t pass the `start` prop, you’ll receive a corresponding error message from the TypeScript compiler.

The `setState` method of the class automatically assumes the appropriate state type, so you don’t need to explicitly specify the type here. Aside from that, there are no changes compared to the class components in JavaScript.

7.6 Summary

This chapter has shown you that a type system such as TypeScript is a good complement to React and its entire ecosystem:

- By using a type system, you can improve the quality and readability of your source code. This is especially true for more extensive applications.
- An alternative to the widely used TypeScript is Flow, developed by Facebook.

- Unlike TypeScript, Flow is not a transpiler that translates source code from one language to another, but a pure type system.
- A *type system* provides you with a basic set of types and allows you to define your own types as classes, interfaces, or type aliases.
- You can specify types in the variable declaration and in the signature of functions so that the TypeScript compiler can ensure compliance with the interface.
- In many cases, you can avoid explicitly specifying types because TypeScript uses its type inference feature to try to determine the appropriate type. However, you should always try to be as explicit as possible, even if that means that you have to write more source code. TypeScript helps you to avoid careless mistakes in this case.
- DefinitelyTyped provides you with type definitions for most third-party libraries that can be installed as NPM packages.
- You provide function components with a typed parameter list as well as a return type.
- React provides type definitions for the Hooks API, so you can specify the appropriate type for the generic `useState` function, for example.
- The Context API also supports type specifications by specifying the type used when calling `createContext`, similar to `useState`.
- For class components, you use TypeScript primarily for structuring the props and the state.

In the next chapter, you'll learn about the options available to you for styling your application and how to combine the onboard tools of React and the features of additional libraries.

Chapter 8

Styling React Components

React is a library for implementing web frontends. In this context, designing the UI is a key issue. However, React doesn't make any specifications here. What initially appears to be a positive aspect turns out to be a disadvantage at second glance. The lack of specifications and guidelines makes it difficult to get started with the implementation of graphical user interfaces.

For you, this means looking at the different approaches and choosing the one that fits your project. In this chapter, we'll introduce several approaches along with their respective advantages and disadvantages and different areas of application so that you can decide which of them you want to use. This will also enable you to evaluate and classify other styling approaches and future solutions yourself.

8.1 CSS Import

You already know the first approach to the styling of components more or less consciously: it consists of using regular CSS stylesheets. For the build process of a React app that you created using Create React App, *Webpack* is used. This tool can include not only JavaScript files, but also other types of files, such as CSS files.

The import statements work in both native JavaScript and TypeScript. The `import` keyword is followed by the name of the CSS file you want to integrate. Make sure that you specify the relative path to ensure the best possible portability of your application. In many React applications, CSS files are given the same names as associated components, but with the `.css` extension. Both files are usually located in the same directory.

An example of such a CSS import is shown in [Listing 8.1](#) in the form of the App component:

```
import ReactElement from 'react';
import './App.css';

const App: React.FC = () => {
  return <h1>Hello React</h1>;
}

export default App;
```

Listing 8.1 CSS Import in the Application (src/App.tsx)

The source code of the *App.css* file consists of regular CSS code. Here you can use all available selectors, properties, and values. A concrete example in the form of the *App.css* file is shown in [Listing 8.2](#):

```
h1 {  
  font-size: 20px;  
  margin: 40px;  
  color: orange;  
  background-color: black;  
}
```

Listing 8.2 Style Specifications for the “App” Component (src/App.css)

8.1.1 Advantages and Disadvantages of CSS Import

The advantage of this way of handling the styling is that it entails very little overhead. It works without any further modifications, and you can use ordinary CSS to style your components. This also ensures support in your development in the form of syntax highlighting and autocompletion.

The disadvantage of this simple variant is that stylesheets do not support namespacing. In the example, the names of the components file and the stylesheet, *App.tsx* and *App.css*, suggest that the two are directly related. However, this impression is deceptive, as the stylesheet is an ordinary global stylesheet. In the example from [Listing 8.2](#), the `h1` selector makes sure that the information affects all `h1` tags. As a rule, however, this is not intended.

You can solve this problem by *namespacing* the styles yourself. The simplest variant is to assign a class name to the root element of a component via the `className` prop and to refer all style specifications to this root element. You can see an example of this in the form of the *BooksList* component in [Listing 8.3](#):

```
import React from 'react';  
import { Book } from './Book';  
import './BooksList.css';  
  
const books: Book[] = [...];  
  
const BooksList: React.FC = () => {  
  return (  
    <table className="BooksList">  
      <thead>  
        <tr>  
          <th>Title</th>  
          <th>Author</th>  
          <th>ISBN</th>
```

```

        <th>Rating</th>
      </tr>
    </thead>
    <tbody>
      {books.map((book) => (
        <tr key={book.id}>
          <td>{book.title}</td>
          <td>{book.author}</td>
          <td>{book.isbn}</td>
          <td>{book.rating} && ↩
            <span>{'.'.repeat(book.rating)}</span></td>
        </tr>
      ))}
    </tbody>
  </table>
);
}

```

```
export default BooksList;
```

Listing 8.3 Namespacing of Styles in a Component with Class Names (src/BooksList.tsx)

The associated stylesheet, which provides a slightly more appealing styling of the table, looks like the one shown in [Listing 8.4](#):

```

table.BooksList {
  border-collapse: collapse;
}

.BooksList th {
  border-bottom: 3px solid black;
}

.BooksList tr:nth-child(2n) {
  background-color: #ddd;
}

.BooksList td {
  padding: 5px 10px;
}

```

Listing 8.4 Styling a Component Using Stylesheets (src/BooksList.css)

When you use stylesheets, you can use various CSS selectors to locate the elements to which each style should be applied. The three basic selectors are *tag*, *class*, and *ID selectors*, although you should avoid using the ID selector. The HTML standard specifies that

an ID must be unique on the current page. However, in the component-based approach of React, which is heavily based on component reusability, there is no way to ensure that there is only one instance of a component. This would also make the ID used no longer unique and invalidate the HTML structure of your application. So you're left with the tag selectors and the class selectors, the latter being used much more frequently as they allow finer control of the styles applied.

8.1.2 Dealing with Class Names

The `class` attribute isn't allowed in JSX because it overlaps with the `class` keyword. JSX is a syntax extension of JavaScript, and the `class` keyword is already used there. Although it would be possible to support the `class` attribute, some features wouldn't work, such as destructuring the `class` prop.

Instead, you must use the `className` attribute. The value of this prop is a string. You can use multiple class names, as in standard HTML, and list as many as you like. Especially when it comes to applying classes based on certain conditions, it gets a bit more elaborate with a string.

As an example of handling multiple class names, you can now implement highlighting of book records that are particularly well-ranked. The table rows of books with a rating of 5 get a yellow background. For this feature to work, you need to extend the `BooksList` component and add the `Highlight` class name to the table rows depending on the rating. The source code of the `BooksList` component with the highlight extension is shown in [Listing 8.5](#):

```
import React from 'react';
import { Book } from './Book';
import './BooksList.css';

const books: Book[] = [...];

const BooksList: React.FC = () => {
  return (
    <table className="BooksList">
      <thead>
        <tr>
          <th>Title</th>
          <th>Author</th>
          <th>ISBN</th>
          <th>Rating</th>
        </tr>
      </thead>
      <tbody>
        {books.map((book) => {
```

```

    const classNames = [];
    if (book.rating === 5) {
      classNames.push('Highlight');
    }
    return (
      <tr key={book.id} className={classNames.join(' ')}>
        <td>{book.title}</td>
        <td>{book.author}</td>
        <td>{book.isbn}</td>
        <td>{book.rating} && ←
          <span>{'*'.repeat(book.rating)}</span></td>
      </tr>
    );
  }}
</tbody>
</table>
);
};

export default BooksList;

```

Listing 8.5 Dynamic Use of Class Names in a Component (src/BooksList.tsx)

In the callback function of the `map` method, which is responsible for displaying the individual lines, you declare the `classNames` constant and initialize it with an empty array. If the `rating` property of the data record has the value 5, you add the `Highlight` entry to the array. In the `tr` element, you define the `className` prop and join the individual class names into a string using the `join` method. This way you can support as many class names as you want, and it doesn't matter if you assign none, one, or multiple class names.

In the component's stylesheet, you add another entry that handles giving the lines with the `Highlight` class a yellow background. In such a case, you must make sure that your specification is more specific than the previous ones; otherwise, the gray background would overwrite the yellow one.

Note

The browser applies the CSS rules based on their *specificity*. For example, a general tag selector is less specific than a class selector, which in turn is less specific than an ID selector. To increase the specificity, you can also combine selectors—for example, a tag selector and a class selector. The combination of parent and child elements also increases the specificity of a selector.

```
table.BooksList {  
  border-collapse: collapse;  
}  
  
.BooksList th {  
  border-bottom: 3px solid black;  
}  
  
.BooksList tr:nth-child(2n) {  
  background-color: #ddd;  
}  
  
.BooksList tr.Highlight {  
  background-color: yellow;  
}  
  
.BooksList td {  
  padding: 5px 10px;  
}
```

Listing 8.6 Styling of the “Highlight” Class (src/BooksList.css)

8.1.3 Improved Handling of Class Names via the “classnames” Library

You don’t necessarily have to perform the task of merging class names yourself as you can leave this task to an external library. An established solution in this context is the `classnames` library. You can add it to your application using the `npm install classnames` command.

The `classnames` library comes with its own type definitions, so you don’t need to install an additional package for it if you use TypeScript in your application. The `classnames` library provides a function that you can pass various data types to, based on which a list of concrete class names is then calculated. In the simplest case, you use several strings, which are then joined together accordingly.

Alternatively, you can also use objects. The key specifies the class name, and the value indicates whether the class should be active. For this purpose, the value is interpreted as a Boolean value. In this case, the `false` value ensures that the class name isn’t used, while `true` activates the respective class. Applied to the `BooksList` component, this leads to the source code shown in [Listing 8.7](#):

```
import React from 'react';  
import { Book } from './Book';  
import './BooksList.css';  
import classnames from 'classnames';
```

```

const books: Book[] = [...];

const BooksList: React.FC = () => {
  return (
    <table className="BooksList">
      <thead>
        <tr>
          <th>Title</th>
          <th>Author</th>
          <th>ISBN</th>
          <th>Rating</th>
        </tr>
      </thead>
      <tbody>
        {books.map((book) => {
          const classes = classNames({ Highlight: book.rating === 5 });
          return (
            <tr key={book.id} className={classes}>
              <td>{book.title}</td>
              <td>{book.author}</td>
              <td>{book.isbn}</td>
              <td>{book.rating} && ←
                <span>{'*'.repeat(book.rating)}</span></td>
            </tr>
          );
        })}
      </tbody>
    </table>
  );
};

```

```
export default BooksList;
```

Listing 8.7 Using the “classnames” Library (src/BooksList.tsx)

Another feature provided by the `classnames` library is the support for arrays. Here too, class names are allowed as strings and objects with class names as keys and Boolean values as values. These arrays, if they are nested structures, are first put into a flat hierarchy and then applied.

Another tool when dealing with stylesheets is a preprocessor like Sass or Less. These can also be integrated into your React application.

8.1.4 Using Sass as CSS Preprocessor

CSS preprocessors add some useful features to the functionality of CSS. In the case of the Sass preprocessor, these include variables, nested rules, or mixins. The next step is to integrate Sass into your application and use this preprocessor instead of the previous stylesheets.

Before you can write your stylesheets in SCSS, one of the stylesheet languages supported by Sass, you must first install the `sass` package using the `npm install -D sass` command. This is the SCSS preprocessor written in Dart that is compiled into JavaScript so that it can be used in the build process of your application.

In the next example, you style the `BooksList` component using SCSS instead of the CSS import used previously. In the `BooksList` component, you use SCSS variables to bundle the background color in a central location. You also use nesting to make the rules a little clearer.

First you create a new file named `variables.scss`. This file contains the definition of the color variable, which makes sure that the variables can be used in multiple places in your application. The source code of this file is shown in [Listing 8.8](#):

```
$grey: #ddd;
$highlight: yellow;
```

Listing 8.8 Variable Definition in SCSS (`src/variables.scss`)

In the next step, you rename the `BooksList.css` file to `BooksList.scss` and modify the source code as shown in [Listing 8.9](#):

```
@import './variables';

table.BooksList {
  border-collapse: collapse;

  th {
    border-bottom: 3px solid black;
  }

  tr:nth-child(2n) {
    background-color: $grey;
  }

  tr.Highlight {
    background-color: $highlight;
  }

  td {
```

```
padding: 5px 10px;
}
}
```

Listing 8.9 “BooksList” Stylesheet in SCSS (src/BooksList.scss)

As you can see in the source code, you use the `@import` statement to import the variables, which you can access by their names in the stylesheet. Another special feature of the SCSS stylesheet is that here the rules are nested within each other: the `table.BooksList` selector forms a kind of namespace where the rules are applied. This has the advantage that the source code is kept clear and no unwanted side effects are created by inadvertently specifying global styles. If you need to access an outside definition—for example, because you need to extend it or specify it in more detail—you can prefix an internal selector with `&` to refer to the outside definition without having to repeat it.

To enable the new styling, you just need to move the import of the stylesheet in the `BooksList` component from the CSS file to the SCSS file.

Styling your components using stylesheets enables you to keep the styling separate from the structure of your application. However, to implement dynamics in the design, you must create a separate class for each customization and include it appropriately via the `className` prop. A more direct option is to use inline styling.

8.2 Inline Styling

The simplest method of styling components is inline styling. In this case, you insert the CSS specifications directly into an element using the `style` prop. But unlike traditional inline styling in HTML, in React it doesn’t consist of a string, but a JavaScript object that is converted by React into a corresponding style specification. In this context, you don’t have the problem of having to deal with assembling a valid string yourself, as you know from specifying dynamic class names.

But you shouldn’t use inline styles excessively. The reason is that inline styles extend the source code of your components significantly and make it unreadable. In addition, its reusability is then severely limited. You can somewhat invalidate both arguments by using the appropriate conventions: as inline styles are objects, you can swap them out from the component’s code and use the full feature set of React, including JavaScript classes and inheritance—which also somewhat invalidates the second argument, as you can reuse the object structures and extend them as needed.

Another disadvantage of inline styles, which cannot be easily avoided, is that pseudoselectors such as `:hover` cannot be implemented. For this purpose, you need to use additional libraries, such as [Emotion \(Section 8.4\)](#).

However, a solution like the already presented combination of the `className` prop and a preprocessor or the use of an additional library is better and much more flexible.

Nevertheless, inline styles have their *raison d'être*. Especially when it comes to individual and dynamic styles, the use of inline styles can be useful. In [Listing 8.10](#), you extend the `BooksList` component to allow your users to click on a line to highlight it:

```
import React, { useState, CSSProperties } from 'react';
import { Book } from './Book';
import './BooksList.scss';

const books: Book[] = [...];

const BooksList: React.FC = () => {
  const [active, setActive] = useState<number | null>(null);

  return (
    <table className="BooksList">
      <thead>
        <tr>
          <th>Title</th>
          <th>Author</th>
          <th>ISBN</th>
          <th>Rating</th>
        </tr>
      </thead>
      <tbody>
        {books.map((book) => {
          const style: CSSProperties = {};
          if (book.id === active) {
            style.backgroundColor = 'yellow';
          }
          return (
            <tr key={book.id} onClick={() => setActive(book.id)}
              style={style}>
              <td>{book.title}</td>
              <td>{book.author}</td>
              <td>{book.isbn}</td>
              <td>{book.rating} && ↩
                <span>{'*'.repeat(book.rating)}</span></td>
            </tr>
          );
        })}
      </tbody>
    </table>
  );
}
```

```
);
};

export default BooksList;
```

Listing 8.10 Inline Styles (src/BooksList.tsx)

For the functionality, you first define a state in which you keep the active data record. When you click on a row, you set the ID of the data record as the active row. In the loop where you display each row, you define an empty object of type `React.CSSProperties`. This type defines the structure of the `style` prop of React elements and helps you to specify the correct style. Then you check if the current data record is active, and if so, you set the background color in the `style` object to yellow. You use the `style` prop to assign the `style` object to the element.

Because inline styling has a higher weighting than element and class styling, this specification overrides the default styling.

As you can see in the example, inline styles in React are slightly different from ordinary CSS rules. Because styles are specified as objects, it's syntactically not possible—or only possible via a workaround—to use CSS properties, which are usually written with a hyphen, as object keys. For this reason, all properties are written in CamelCase. In the case of the `background-color` property, this results in `backgroundColor`. You specify the values as ordinary JavaScript strings.

When specifying numerical values, such as for height or width, there is another special feature: here, React automatically inserts the `px` unit. If you need a different unit, such as `em`, you can specify the value as a string—that is, as `'35em'`.

Another alternative to styling React components is to use CSS modules.

8.3 CSS Modules

CSS modules represent a feature that's supported by Create React App. This is standards-compliant CSS, but only valid for the current component. In this case, you don't have to bother with namespacing on your own. CSS modules are imported in a component. You can then access the styles defined in the CSS module through an object structure. To import the module correctly, the filename of the CSS module must end in `.module.css`. [Listing 8.11](#) contains the source code of the module for styling the `BooksList` component. You save the source code in a file called `BooksList.module.css`:

```
.BooksList {
  border-collapse: collapse;
}

.header {
```

```
border-bottom: 3px solid black;
}

.tableRow:nth-child(2n) {
  background-color: #ddd;
}

.cell {
  padding: 5px 10px;
}
```

Listing 8.11 CSS Module for the “BooksList” Component (src/BooksList.module.css)

As you can see, the CSS module is standards-compliant CSS, so you can make use of all the tools your development environment offers for CSS. For CSS modules, you mainly use class names. You can also insert tag names, but in most cases the classes should be sufficient.

The CSS module can be integrated into the component by way of importing it. [Listing 8.12](#) contains the necessary source code:

```
import React from 'react';
import { Book } from './Book';
import styles from './BooksList.module.css';

const books: Book[] = [...];

const BooksList: React.FC = () => {
  return (
    <table className={styles.BooksList}>
      <thead className={styles.header}>
        <tr>
          <th>Title</th>
          <th>Author</th>
          <th>ISBN</th>
          <th>Rating</th>
        </tr>
      </thead>
      <tbody>
        {books.map((book) => {
          return (
            <tr key={book.id} className={styles.tableRow}>
              <td className={styles.cell}>{book.title}</td>
              <td className={styles.cell}>{book.author}</td>
              <td className={styles.cell}>{book.isbn}</td>
              <td className={styles.cell}>
```

```

        {book.rating && <span>{'*'.repeat(book.rating)}</span>}
      </td>
    </tr>
  );
  }}}
</tbody>
</table>
);
};

```

```
export default BooksList;
```

Listing 8.12 Integration of the CSS Module into the “BooksList” Component (src/BooksList.tsx)

The build process of Create React App supports the CSS modules directly, and you don't need to do any further configuration. This is also true if you use TypeScript, as in the example. During the build process, the component and CSS module are parsed, and automatically generated class names are inserted in the rendered component. This ensures that there are no name conflicts with other components. The name of the class follows the pattern `<filename>_<classname>__<hash>`. For example, in the `.header` class example, this results in a class name like `BooksList_header__ayrMz`.

As you can see in the source code of the CSS module, pseudoselectors like `hover` or `nth-child` are also supported. To use them, you need to append the selector to the class name as usual.

For simple use cases, the functionality of CSS modules is usually sufficient. But if it isn't, you can combine this feature with other functionalities and libraries if necessary, such as with the `classnames` library.

By using CSS modules, you stay very close to the CSS standard while still having the advantages of isolated styles per component. Moreover, CSS modules additionally support the native pseudoselectors. However, this is not the last approach we'll present for styling your React components. In the following sections, you'll get to know another possibility: the Emotion library.

8.4 CSS in JavaScript Using Emotion

Emotion is a library for *CSS-in-JS*, an approach where you make style specifications in JavaScript code. There are quite a few libraries you can draw on. Probably the best known in this area are *Styled Components* and *Emotion*. *Styled Components* specializes in React, while *Emotion* takes a framework-independent approach. In the next sections, we'll introduce *Emotion* as a possible solution because it provides the same syntax as *Styled Components*, but more features, and you can use the library very well with React.

You can use Emotion either with the `css` prop or with the `styled` approach. The `css` prop allows you to write the styles in a prop and apply them directly in the element. This variant is similar to direct styling via the `style` prop. The `styled` approach follows an entirely different strategy than the ways of styling a component we've presented up to this point. The `styled` approach uses a kind of higher-order component. This exploits a rarely used part of the ECMAScript standard by using the *tag function* of the template strings, a function executed to process the template string.

By default, the JavaScript environment provides a standard tag function that is used to perform variable substitution. Emotion, or which follows the same principles as the idea generator *styled components*, takes this idea further and allows you to create components with local style specifications via template strings. In addition to just styling components, the Styled Components library offers numerous other useful features, some of which we'll take a closer look at ahead and integrate into the example application.

8.4.1 Installing Emotion

Emotion is an open-source library developed under the MIT license on GitHub. Depending on which variant of Emotion you want to use, you need to install different packages. For use in the `css` prop, all you need is the `@emotion/react` package, which you can install using `npm install @emotion/react`. If you want to use the `styled` approach, you have to install the `@emotion/styled` package in addition to that; that is, you need to run the `npm install @emotion/react @emotion/styled` command. Both packages come with their own type definitions, so you don't need to install additional packages to use them with TypeScript.

8.4.2 Using the “css” Prop

The `css` prop of Emotion digs deep into React's build process and makes the Babel plugin, which is responsible for translation, use the `jsx` function instead of the `React.createElement` function. To use this variant of Emotion with Create React App and TypeScript, you need to adjust the TypeScript configuration and add the `"jsxImportSource": "@emotion/react"` entry in the `compilerOptions` of the `tsconfig.json` file. Then you still need to add a comment at the beginning of the file so that Babel processes it correctly. [Listing 8.13](#) shows an example of using the `css` prop of Emotion:

```
/** @jsxRuntime classic */
/** @jsx jsx */
import { jsx, css } from '@emotion/react';
import React from 'react';
import { Book } from './Book';

const headerStyle = {
  borderBottom: '3px solid black',
};
```

```

const cellStyle = {
  padding: '5px 10px',
};

const books: Book[] = [...];

const BooksList: React.FC = () => {
  return (
    <table css={{ borderCollapse: 'collapse' }}>
      <thead css={headerStyle}>
        <tr>
          <th>Title</th>
          <th>Author</th>
          <th>ISBN</th>
          <th>Rating</th>
        </tr>
      </thead>
      <tbody>
        {books.map((book) => {
          return (
            <tr
              key={book.id}
              css={css`
                &:nth-of-type(2n) {
                  background-color: #ddd;
                }
              `}
            >
              <td css={cellStyle}>{book.title}</td>
              <td css={cellStyle}>{book.author}</td>
              <td css={cellStyle}>{book.isbn}</td>
              <td css={cellStyle}>
                {book.rating} && <span>{'.'.repeat(book.rating)}</span>
              </td>
            </tr>
          );
        })}
      </tbody>
    </table>
  );
};

export default BooksList;

```

Listing 8.13 Use of the “css” Prop of Emotion (src/BooksList.tsx)

The entry point to the file consists of two comments that Emotion and the build process respectively need to handle the `css` prop correctly. The first comment switches the runtime of the `preset-react` plugin to classic mode. The second comment makes sure that the `css` prop is processed correctly by Emotion. If your development environment has problems with the correct processing of the `css` prop, you can fix it with the line:

```
/// <reference types="@emotion/react/types/css-prop" />
```

Before you get into the component, you define two style objects. The `headerStyle` object is responsible for the appearance of the table's header row, while `cellStyle` is responsible for the appearance of the table's individual cells.

In the `BooksList` component itself, you can see different variants of using the `css` prop:

- **Inline object**

In the `table` element, you define the `css` prop and set the styling directly inline. On the one hand, this has the advantage that you can see at a glance which styles are defined for an element. If you use this approach for multiple elements and have multiple style specifications per element, the source code of your component quickly becomes confusing. You should therefore only use this variant sparingly.

- **External object**

For the table header and the individual cells, you swap out the styles to separate objects so that the style definitions do not appear directly in the component. Using this variant, you achieve a tidier source code in the component and can reuse the individual objects in several places, as you can see with the `td` elements. Also, you can swap out these style objects to a separate file if needed.

- **CSS template string**

In the third variant, you define a template string with the styles and provide it with the `css` function of Emotion. With this approach, you can use pseudoselectors such as `nth-of-type` or `hover`, for example. The same approach applies to the template string as to an object: you can define it either inline as in the example or outside the component.

Emotion offers a second variant—the styled approach—to style your components.

8.4.3 The Styled Approach of Emotion

The *styled approach*, also referred to as *Emotion styled components*, is a styling variant in which you use template strings to produce new components. Emotion is not the first library to implement this approach. The solution was originally derived from libraries like `Styled Components` and `Glamorous`.

The advantage of styled components over the `css` prop is that you don't need to make any adjustments to your TypeScript configuration and your component, especially when using TypeScript. To be able to use styled components, you just need to install the `@emotion/styled` package.

There are several approaches to organizing styled components. These range from placing the components directly in the file where they're needed to having a separate file per Styled Component. The first approach has the disadvantage that styled components are relatively tightly coupled to the component in which they are used. The second approach results in a very large number of files that may make your application very confusing. A good middle ground is to group styled components thematically into files.

For the sample application, you create a new file named *BooksList.styles.ts*. The styled components of Emotion provide tagging capabilities for all HTML elements, which you can use to create corresponding elements with style definitions for your application. You can directly convert the static styles of the table for the table element, the table header, and the cells. The syntax is similar to the CSS template strings you learned about in the context of the `css` prop. [Listing 8.14](#) shows the implementation of the components as styled components:

```
import styled from '@emotion/styled';

export const Table = styled.table`
  border-collapse: collapse;
`;

export const THead = styled.thead`
  border-bottom: 3px solid black;
`;

export const TD = styled.td`
  padding: 5px 10px;
`;
```

Listing 8.14 Creation of Styled Components (src/BooksList.styles.ts)

To create the styled components, you can use the `styled.table`, `styled.thead`, and `styled.td` tag functions. You pass the CSS information to them in the template string.

Because styled components are JavaScript strings and not regular CSS, you have no direct support in your development environment. However, a plugin is available for both WebStorm and Visual Studio that gives you syntax highlighting and autocompletion for styled components.

The result of the tag function is a full-fledged React component that you can use within your application:

```
import React from 'react';
import { Book } from './Book';
import { Table, TD, THead } from './BooksList.styles';

const books: Book[] = [...];
```



```
const BooksList: React.FC = () => {
  return (
    <Table>
      <THead>
        <tr>
          <th>Title</th>
          <th>Author</th>
          <th>ISBN</th>
          <th>Rating</th>
        </tr>
      </THead>
      <tbody>
        {books.map((book) => {
          return (
            <tr key={book.id}>
              <TD>{book.title}</TD>
              <TD>{book.author}</TD>
              <TD>{book.isbn}</TD>
              <TD>{book.rating} && ←
                <span>{'*'.repeat(book.rating)}</span></TD>
            </tr>
          );
        })}
      </tbody>
    </Table>
  );
};

export default BooksList;
```

Listing 8.15 Integration of the Styled Component (src/BooksList.tsx)

As you can see in [Listing 8.15](#), you can choose the names of your styled components in such a way that, except for the initial uppercase letter, you won't notice any difference from the React elements you've been using so far. However, the styled components of Emotion can do much more for you and your application.

8.4.4 Pseudoselectors in Styled Components

Up to this point, you've used pseudoselectors to ensure that the individual rows of the table were colored differently. You can also achieve this with styled components by using such selectors in the template string. [Listing 8.16](#) contains the implementation of the TR component that takes care of the coloring of the table rows:

```
export const TR = styled.tr`
  &:nth-of-type(2n) {
    background-color: #ddd;
  }
`;
```

Listing 8.16 Pseudoselectors in Styled Components (src/BooksList.styles.ts)

Instead of the `nth-child` selector, you use the `nth-of-type` selector in this example. The reason is that `nth-of-type` produces the same result as `nth-child`, but the former is the better variant for server-side rendering. For this reason, you should generally use this selector. If you use `nth-child`, you will receive a corresponding warning in the developer tools console of your browser advising you against using this selector.

To activate the new component, you need to import it into the `BooksList` component and replace the `tr` element in the body of the table with the `TR` styled component.

8.4.5 Dynamic Styling

You can also make your styled components dependent on props. This further increases the flexibility of your components and allows you to support multiple variants. To demonstrate a styled component that can be controlled by props, you allow your users to highlight an entry in the table by clicking on a row. The `BooksList` component then sets the `highlight` prop for the respective line, and the `TR` component takes care of setting the correct background color.

```
import { css } from '@emotion/react';
import styled from '@emotion/styled';

...
```

```
type TRProps = {
  highlight: boolean;
};
```

```
export const TR = styled.tr`
  &:nth-of-type(2n) {
    background-color: #ddd;
  }
  ${({ highlight }: TRProps) =>
    highlight &&
    css`
      &&& {
        background-color: yellow;
      }
    `
  }
`;
```

```
    }  
  }  
};
```

Listing 8.17 Styling of Styled Components, Depending on Props (src/BooksList.styles.ts)

To support the `highlight` prop, you first define a type for the component props that contains the `highlight` property of Boolean type. Then you can define an arrow function in the template string of the styled component. By default, this arrow function receives the props you pass to the component. Depending on whether the `highlight` prop is set and contains the value `true`, you output another template string from the `@emotion/react` package using the `css` function. The `&&&` selector allows you to ensure that the subsequent `background-color` specification is given a higher priority than the `nth-of-type` selector; otherwise you won't be able to highlight the rows colored by it.

Based on this customization you can integrate the new prop into the `BooksList` component. The corresponding source code is shown in [Listing 8.18](#):

```
import React, { useState } from 'react';  
import { Book } from './Book';  
import { Table, TD, THead, TR } from './BooksList.styles';  
  
const books: Book[] = [...];  
  
const BooksList: React.FC = () => {  
  const [active, setActive] = useState<number | null>(null);  
  
  return (  
    <Table>  
      <THead>  
        <tr>  
          <th>Title</th>  
          <th>Author</th>  
          <th>ISBN</th>  
          <th>Rating</th>  
        </tr>  
      </THead>  
      <tbody>  
        {books.map((book) => {  
          return (  
            <TR  
              key={book.id}  
              onClick={() => setActive(book.id)}  
              highlight={active === book.id}  
            >
```

```

        <TD>{book.title}</TD>
        <TD>{book.author}</TD>
        <TD>{book.isbn}</TD>
        <TD>{book.rating && ←
          <span>{'?'.repeat(book.rating)}</span>}</TD>
      </TR>
    );
  }}
</tbody>
</Table>
);
};

```

```
export default BooksList;
```

Listing 8.18 Controlling a Styled Component via Props (src/BooksList.tsx)

The changes to the `BooksList` component are limited to the state in which you store the information about which row is highlighted, and to the `TR` component itself. Here you define a click handler that marks the current row as the active row and the `highlight` prop that contains the value `true` or `false` depending on the currently active row.

8.4.6 Other Features of Styled Components

Styled components offer a comparatively large range of functions. For example, in addition to the features already described, it's also possible to add additional styles to existing components. For this purpose, you can pass a component to the `styled` function and then define the additional styles via a template string. In [Listing 8.19](#), you can see this with the example of a table cell:

```

const BaseTd = styled.td`
  padding: 5px 10px;
`;

export const TD = styled(BaseTd)`
  border: 1px solid black;
`;

```

Listing 8.19 Extension of a Styled Component (src/BooksList.styles.ts)

In the example, you first define a `BaseTd` component that you create using `styled.td`. This component already contains a basic style with the `padding` specification. Then you call the `styled` function with the `BaseTd` component, and you can define more styles. This approach becomes particularly interesting when you use it to implement style inheritance.

As an alternative to the approaches presented so far, you can also include pure CSS solutions in your React application, such as Tailwind.

8.5 Tailwind

Tailwind is a CSS framework that you can use independently of JavaScript frameworks and libraries. Tailwind is a bit of an acquired taste when it comes to styling DOM structures. As a utility-first framework, it provides you with a variety of classes that you can combine in your elements to achieve the styling you want.

This approach initially results in your elements getting a long list of class names, which can make the code a bit confusing. However, the component-based approach of React makes sure that class repetition is kept in check because you define standard elements, such as buttons, only once per application. The big advantage of Tailwind is that after a certain point the amount of CSS code does not grow or grows very little, because you have included all the classes you need for your application and only reuse them from that point on.

To integrate Tailwind, you must first install the `tailwindcss`, `postcss`, and `autoprefixer` packages using the `npm install --save-dev tailwindcss postcss autoprefixer` command. The `postcss` and `autoprefixer` packages are CSS transformers that make sure that your CSS code is valid and executable in as many browsers as possible.

Then you create the configuration files for Tailwind (*tailwind.config.js*) and PostCSS (*postcss.config.js*) using the `npx tailwindcss init -p` command. For use in your React application, you want to customize the Tailwind configuration as shown in [Listing 8.20](#) by adding the path to your components to the `content` array:

```
module.exports = {
  content: ['./src/**/*.{js,jsx,ts,tsx}'],
  theme: {
    extend: {},
  },
  plugins: [],
};
```

Listing 8.20 Customization of the Tailwind Configuration (*tailwind.config.js*)

After that, you still need to integrate Tailwind into your application. You can do this by adding the `@tailwind` directives to your *index.css* file, as shown in [Listing 8.21](#):

```
@tailwind base;
@tailwind components;
@tailwind utilities;
```

Listing 8.21 Integration of Tailwind (*src/index.css*)

With these customizations, you can use Tailwind in your application. You can see a concrete example of this in [Listing 8.22](#) with the `BooksList` component:

```
import React from 'react';
import { Book } from './Book';

const books: Book[] = [...];

const BooksList: React.FC = () => {
  return (
    <table className="border-collapse">
      <thead className="border-b-4 border-black">
        <tr>
          <th>Title</th>
          <th>Author</th>
          <th>ISBN</th>
          <th>Rating</th>
        </tr>
      </thead>
      <tbody>
        {books.map((book) => {
          return (
            <tr key={book.id} className="even:bg-gray-300">
              <td className="py-1 px-2">{book.title}</td>
              <td className="py-1 px-2">{book.author}</td>
              <td className="py-1 px-2">{book.isbn}</td>
              <td className="py-1 px-2">
                {book.rating} && <span>{'*'.repeat(book.rating)}</span>
              </td>
            </tr>
          );
        })}
      </tbody>
    </table>
  );
};

export default BooksList;
```

Listing 8.22 Tailwind Styling of the “BooksList” Component (src/BooksList.tsx)

At first glance, Tailwind classes are reminiscent of inline styling because you use a separate class for almost every piece of style information. The advantage of Tailwind is that you can choose from a limited set of default classes. Tailwind does allow you to

define your own classes, but you can usually go very far with the default classes. These classes force you always to use the same specifications, leading to good reusability.

For the `thead` element, the `border-b-4` class provides a four-pixel-wide border at the bottom edge of the element. The `border-black` class colors the frame black. You can color the `tr` elements using the `even:bg-gray-300` class. This ensures that every other element gets a gray background. Finally, with cells, you can see how Tailwind allows you to work with spacing. The `py-1` class stands for padding at the top and bottom with a size of four pixels. You can configure the size of the distance in Tailwind. By default, the 1 represents 0.25 rem or four pixels. A 2 correspondingly means 0.5 rem or eight pixels.

Tailwind has many more features, such as responsiveness, a configurable theme, or extensions of the framework through functions and directives. The combination of Tailwind and React allows you to apply styles to components for a high degree of reusability.

8.6 Summary

In this chapter, you learned about different approaches that you can use to style the components of your application:

- The simplest solution consists of importing CSS files. This solution has the disadvantage that the style specifications are globally valid and can have unwanted side effects.
- The `classnames` library makes it easier to handle the `classname` attribute and allows you to dynamically manage the list of class names you want to apply to an element.
- If you install the `sass` package, you can use SCSS in your application instead of CSS and have access to advanced features like nesting, variables, and mixins.
- *Inline styles* allow you to apply styles directly to the elements of your components. By using JavaScript objects, you have great flexibility here and can quickly apply custom styles. The biggest disadvantage of this variant is the nonexistent reusability of the styles.
- *CSS modules* enable you to write standards-compliant CSS that affects only the component in question.
- The *CSS-in-JS* approach takes the idea of inline styles and tries to eliminate their major drawbacks. One concrete implementation of CSS-in-JS is the Emotion library.
- Using Emotion, you can apply styles directly to elements using the `css` prop. Here you can use both objects and template strings.
- The *styled components* of Emotion let you create style definitions using template strings to create wrapper components. This approach has the advantage that you don't need to interfere with the build process of React.

- The *Tailwind* CSS framework takes a different approach and allows you to define the styles of your components with a specified set of utility classes. You can customize Tailwind to a high degree and adapt it exactly to your needs.

The next chapter is about testing your application using automated tests. In this context, you'll learn which tools are available to you and how you can test the various components of your application.

Chapter 9

Securing a React Application through Testing

The more extensive an application becomes and the more people work on it, the greater the risk that something will go wrong and break. For you, this means that there is always an uneasy feeling associated with major modifications to your code: Will the complete application still work after the integration of the new feature or after the completion of the rebuild? To eliminate this uncertainty as far as possible and reduce manual testing effort, you can use automated tests.

In addition to the metric of how many of the existing tests fail, another rather inconspicuous metric is very important: How long does it take to write a new test? This metric is relatively low in React thanks to numerous helper constructs. In this chapter, you'll learn how to write tests using the Jest testing framework, what to look for when dealing with dependencies, and what snapshot tests are all about. You'll also learn about some libraries and packages that can support your testing efforts.

But before we get into the real topic of this chapter, let's first look at testing in general and discuss why it's worth writing tests for your application.

First and foremost, tests reduce the effort of manual testing. Automated testing provides several benefits for application development:

- **Security**

Because tests cover not only the current feature or module, but large parts of the application, negative side effects can be detected and corrected early on, so you will no longer be surprised by errors.

- **Documentation**

By carrying out your tests, you document how the interfaces and components of your application should be used. Unlike source code documentation, which can easily become outdated, outdated tests will fail.

- **Fast and direct feedback**

When you embed the generation of tests into your development process, the results of the tests give you very fast and direct feedback. Feedback is particularly helpful in test-driven development.

When creating tests, you should consider some basic aspects:

- The environment of your application should be prepared in such a way that you can start writing a test directly without having to solve configuration issues first.

- The individual tests as well as the entirety of all tests must run very quickly. Of course, this time span can vary depending on the number and scope of the tests—anything between a few seconds and a few minutes is still acceptable. If the tests run longer, you won't receive feedback immediately. You either have to wait for the tests or let them run independently of the development process. In addition, tests that run for a long time tend not to be run as frequently as those that run for a very short time.
- Tests should be independent of other systems. If your tests are dependent on other systems such as web services or databases, this immediately creates several disadvantages: If one of the systems fails, the tests will automatically fail as well. The simulation of error or exception cases usually requires a manipulation of the third-party system. Finally, dependent tests have a significantly extended runtime due to the latency in communication.
- The execution of the tests must not require manual user interaction. This means that the tests should be fully automated. Only when this condition is met can they be integrated into a continuous integration system and be automatically executed each time source code gets released.
- Tests should be run in a clean environment and leave an equally clean environment once they are done. Through setup and teardown routines, you should make sure that the testing environment is always clean and that tests don't fail due to side effects.
- The individual tests of your application must not build on each other. Once the order of the tests determines success or failure, you have a tight coupling between the tests. If one test in the series fails, there is a high risk that subsequent tests will also fail, even though the functionality being tested may be error-free. Tests that depend on each other also do not allow for a random execution order or the execution of individual tests due to the dependency.

9.1 Getting Started with Jest

While React doesn't dictate anything when it comes to testing frameworks, using Jest is the most obvious choice. The testing framework, like the package manager Yarn, is developed by Facebook. Alternatives to Jest include Jasmine and Mocha, and there are very good step-by-step tutorials available for both frameworks. In this chapter, we'll focus entirely on testing with Jest as this is the most common combination. In addition, Create React App already has everything ready for you, so you can start developing tests in Jest right away.

As with many other tools, the development team took existing solutions as a model and fixed their weaknesses as much as possible, so you can find numerous design patterns and syntactic elements from other testing frameworks in Jest as well. If you already have experience using Jasmine or Mocha, switching to Jest will be easy.

Jest: The Testing Framework from Facebook

Jest provides an elegant and flexible solution for the automated testing of a React application. Unlike other frameworks, it doesn't require any separate infrastructure. In the case of Jasmine, for example, you need infrastructure components such as Karma, which provides you with the browser environment in which to run the tests.

Jest takes a different approach as it's based on a library called `jsdom`, which simulates the browser environment. Here, the development team makes a compromise and accepts that the testing environment may behave differently from a real browser in some details. But you gain some distinct advantages: The runtime of the tests is significantly reduced because the overhead of a full-fledged browser is eliminated. And by eliminating the browser instance, both the infrastructure maintenance and runtime are significantly reduced.

Although Jest is very commonly used in conjunction with React, it can also be used in other environments, such as on the server side with Node.js or with other frontend frameworks like Angular.

9.1.1 Installation and Execution

If you use Create React App, Jest is already preinstalled. If you need to install the framework, you can do so via the `npm install --save-dev jest` command. Jest can be executed using the `npx jest` command. In the simplest case, Jest doesn't require a separate configuration.

The testing framework finds tests by their naming convention in the file system. The names of the test files end in `.spec.js` and `.spec.ts` respectively. Once you've created your application with Create React App, the `react-scripts` package is used to run the tests. For this, there is an entry in the `package.json` file of your application that executes the `react-scripts` command with the `test` option.

In this type of execution, Jest is in watch mode and automatically runs the tests whenever a change is made in the source code. Here you have the possibility to influence the behavior of Jest through different commands. An overview of this is shown in [Table 9.1](#).

Command	Meaning
<code>[Shift]</code>	Run all tests
<code>[f]</code>	Run only failed tests
<code>[q]</code>	Exit watch mode
<code>[i]</code>	Run failed tests in interactive mode
<code>[p]</code>	Filter file names according to a pattern

Table 9.1 Commands for the Jest CLI

Command	Meaning
<code>t</code>	Filter by test name
<code>Enter</code>	Trigger test run

Table 9.1 Commands for the Jest CLI (Cont.)

For improved performance, Jest is capable of running multiple tests simultaneously. Also, in watch mode, only those tests where the source code has changed are run. This applies to both the source code of the test and the code that the test is supposed to check.

9.1.2 Organization of the Tests

Where an application's tests are placed in the file system depends on several factors. First, it's crucial what kind of tests we're talking about in the first place. *Integration or end-to-end (E2E) tests* test complete features. In most cases, these tests have no direct counterpart in the file system or involve several files, so a direct assignment is difficult. For this reason, E2E tests are usually managed in a separate directory.

For *unit tests*, the situation is different. These tests check components or functions, which means that a test can be assigned to a file. This allows you to place the tests either directly with the files you want to test, or in a separate directory structure that corresponds to the directory structure of the application. Usually the first approach is followed in React applications; that is, the test is managed in the same directory as the production code. Due to their different file extension, the tests can be easily distinguished from the other files. The assignment is made via the remaining file name.

9.1.3 Jest: The Basic Principles

Before you start writing tests for your application, let's first look at the features of Jest and how you can write tests using the framework. For this purpose, you implement a simple calculator class that provides methods using the four basic arithmetic operations.

The basis for the following examples is an empty application that you created using Create React App. In the first step, you create two files in the *src* directory. The first file is named *calculator.ts* and contains the actual source code. The second file is named *calculator.spec.ts* and contains the tests. When you execute the `npm test` command on the console, you'll receive the following message: Your test suite must contain at least one test. This means that no tests were found that can be executed. The reason is that you have created a test file but haven't yet written a test, which we'll change in the next step.

Related Topic: Test-Driven Development

Test-driven development (TDD) is a way of developing software in which you proceed in cycles. A *cycle* consists of three steps: red, green, and refactor.

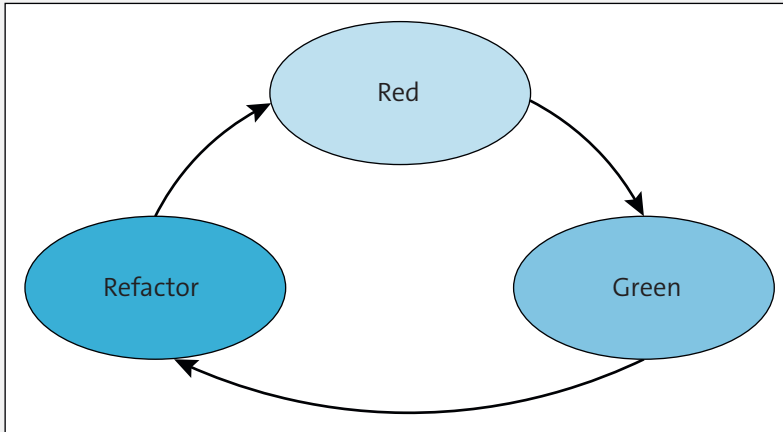


Figure 9.1 TDD cycle

■ Red

The TDD cycle always starts with a test. In this test, you formulate your expectations for a part of your application. Because no implementation exists at this point, the test fails as expected. A failure is marked with the color red by numerous testing frameworks, hence the name of this step.

■ Green

The goal in the second step is to produce a positive testing result. This doesn't necessarily mean that this is the final solution. The only important thing here at first is that the test passes successfully. A positive testing result is indicated by a green mark in the output, which is why this step is called *green*.

■ Refactor

The final step in the cycle is the rebuilding of the existing source code. Does the source code not yet comply with the coding guidelines, or can the implemented algorithm still be improved? Then now is the right time to improve the source code. The important thing here is that the test continues to run successfully. However, the refactor step is not limited to the source code of your application. It's also allowed to improve the code of your tests. Classics here include the reduction of duplicates or the general improvement of the code style.

If you strictly follow TDD, you do not write untested source code. In addition, you do not generate unnecessary source code because you formulate your requirements for the application as tests and write only source code that is used to meet those requirements.

In TDD, you will occasionally come to a point where you need to consider what test you want to write next. The answer to this is relatively simple: write a test that will give you more insight and information about your implementation. On the other hand, this means that you should stop formulating tests if these tests do not provide any further information.

Take, for example, the addition operation in our example. If you have written a test for $1 + 1$ and $1 + 2$, another test to check whether the result is also correct for $2 + 2$ doesn't really add any value. Instead, you should consider which test will help you. For example, you could write a test that ensures that no letters can be passed.

9.1.4 Structure of a Test: Triple A

You can formulate a test in Jest using either the `test` or the `it` function. Both functions are equivalent, but we use the `it` function here because it's also common in other frameworks like Mocha or Jasmine.

The first argument you pass to this function is a short description of what the test checks. This string gets output by Jest in case of an error. So it should be descriptive enough to allow you to draw a conclusion about the issue at first glance, and short enough that you don't have to read an entire paragraph to understand the context. If you can't describe a test in a short sentence, or if you use the word "and" too often to connect multiple cases, then this is a clear indication that the test is doing too many things at once and that you're better off writing several smaller tests instead.

You need to formulate the actual test in a callback function, which you pass as the second argument to the `it` function. A classic unit test always follows the same structure, which can be described by the triple A pattern: *arrange*, *act*, and *assert*.

Arrange: Preparing the Environment

The first step is to prepare the testing environment. In the running example, this means that you create an instance of the `calculator` class. For example, when you test a React component, this would mean that you need to create an instance of the component in order to interact with it.

In the example, you store the instance of the `calculator` class in a variable that you can reference later in the test.

Act: Performing the Action

The term *act* stands for the execution of the action to be tested. In a classic unit test, you execute the function you want to test (in this case, the `add` method) and get a return value, which you cache in a variable.

Assert: Checking the Result

The last part of the test is the most exciting one. Here you check if the result meets your expectations. For this purpose, Jest provides you with a set of matchers that you can use to formulate your condition. The structure of the assert statement is always as follows: `expect([result]).[matcher]([expectation])`. In the concrete example, the assertion is `expect(result).toBe(2)`.

The main reason for splitting the test into three parts is that it makes the source code clearer and helps the people reading your source code to find their way around better and faster. In addition, caching the individual steps in variables makes debugging the test much easier. Finally, this split also allows you to eliminate duplicates in your test code more easily—but more on that later. In [Listing 9.1](#), you can find the source code of the test, and [Listing 9.2](#) contains the corresponding implementation.

```
import Calculator from './calculator';

it('should add 1 and 1 and return 2', () => {
  const calculator = new Calculator();
  const result = calculator.add(1, 1);
  expect(result).toBe(2);
});
```

Listing 9.1 First Test with Jest (src/calculator.spec.ts)

```
export default class Calculator {
  add(a: number, b: number) {
    return a + b;
  }
}
```

Listing 9.2 Implementation for the Test (src/calculator.ts)

If you've run `npm test` on the console, Jest will automatically run the tests when you make changes to the files, and you'll get a success message as the result.

9.1.5 The Matchers of Jest

The `toBe` matcher used so far represents the simplest variant. However, Jest offers several other matchers that you can use in your tests. [Table 9.2](#) contains an overview of some of them.

In addition to the matchers presented in [Table 9.2](#), Jest, like many other testing frameworks, also allows you to define custom matchers. In practice, however, it's rather rare that the available matchers are not sufficient.

Matcher	Meaning
toBe	Checks if the result matches the passed value. Here, <code>Object.is</code> is used, which checks for reference equality for objects.
toEqual	This matcher checks if the two structures to be compared match. For this purpose, iteration is performed over each element.
toBeTruthy/toBeFalsy	You can use these matchers to check whether they are <i>truthy</i> or <i>falsy</i> values. A <i>truthy value</i> can be cast to the value <code>true</code> in JavaScript, and a <i>falsy value</i> can be cast to <code>false</code> accordingly.
toBeDefined/toBeUndefined	The <code>toBeUndefined</code> matcher checks if the passed value is undefined; the <code>toBeDefined</code> matcher checks if the value is not undefined.
toBeNull	You can use this matcher to determine if the value <code>null</code> was passed.
toBeGreaterThan/ toBeGreaterThanOrEqual/ toBeLessThan/ toBeLessThanOrEqual	These matchers allow you to check, for numerical values, whether a number is larger or smaller than another number.
toBeCloseTo	For floating point operations like <code>0.1 + 0.2</code> , the result in many languages is not exactly <code>0.3</code> , but slightly larger than <code>0.3</code> . This matcher enables you to check if the result is close to <code>0.3</code> .
toMatch	This matcher lets you compare a string with a regular expression.
toContain	You can use this matcher to check if a certain element is contained in an array.

Table 9.2 Matchers in Jest

Sometimes it happens that you want to check exactly the opposite of what a matcher offers you. To avoid having to negate the result or expectation now, which on the one hand is error-prone and on the other hand worsens the readability of the source code, you can insert a `.not` before of the matcher, which reverses its meaning. For example, `expect(result).not.toBe(2)` would result in the `result` variable being allowed to contain any value except 2.

9.1.6 Grouping Tests: Test Suites

In this chapter, you've already come across *test suites*, groups of tests, even if this was only in the form of a message from Jest. Jest allows you to group tests so that you can better arrange thematically related tests. These groups are relevant because you can provide each group with a short description, similar to the description of a test. In addition, you can skip or run an entire test suite exclusively, and setup and teardown routines also work with test suites.

You can create a test suite using the `describe` function. As already mentioned, this function accepts a short description of the test suite and a callback function, within which you can formulate your tests. In addition to tests, Jest allows you to arrange other test suites within a test suite. For the `calculator` example, a structure might look like [Listing 9.3](#):

```
import Calculator from './calculator';

describe('Calculator', () => {
  describe('add', () => {
    it('should add 1 and 1 and return 2', () => {
      const calculator = new Calculator();
      const result = calculator.add(1, 1);
      expect(result).toBe(2);
    });
  });
});
```

Listing 9.3 Test Suites in Jest (`src/calculator.spec.ts`)

In this example, the class gets a test suite, while the `add` method in turn gets a child test suite that contains all the tests that affect the `add` method.

9.1.7 Setup and Teardown Routines

If you integrate a second test in the next step, which for example checks whether $-1 + 1$ results in the value 0, this leads to the source code shown in [Listing 9.4](#):

```
import Calculator from './calculator';

describe('Calculator', () => {
  describe('add', () => {
    it('should add 1 and 1 and return 2', () => {
      const calculator = new Calculator();
      const result = calculator.add(1, 1);
      expect(result).toBe(2);
    });
  });
});
```

```
it('should add -1 and 1 and return 0', () => {  
  const calculator = new Calculator();  
  const result = calculator.add(-1, 1);  
  expect(result).toBe(0);  
});  
});  
});
```

Listing 9.4 Another Test for the Addition Operation (src/calculator.spec.ts)

You can obtain these tests by duplicating the code of the existing test and adapting it accordingly. Such operations in particular lead very quickly to large amounts of duplicates in the code. At this point, it's necessary to eliminate them in a targeted manner.

If you look at the three steps of the test, you'll notice that the arrange step has been taken over unchanged. The other two—act and assert—have at least different values. In such a case, where the initialization of the environment includes the same steps for each test, you can outsource it to a so-called setup routine. These functions are registered along with the `beforeEach` function and executed before each test in the current test suite. This will ensure that a new and clean environment is created for each test to run in. [Listing 9.5](#) shows the appropriate source code:

```
import Calculator from './calculator';  
  
describe('Calculator', () => {  
  describe('add', () => {  
    let calculator: Calculator;  
    beforeEach(() => {  
      calculator = new Calculator();  
    });  
  
    it('should add 1 and 1 and return 2', () => {  
      const result = calculator.add(1, 1);  
      expect(result).toBe(2);  
    });  
  
    it('should add -1 and 1 and return 0', () => {  
      const result = calculator.add(-1, 1);  
      expect(result).toBe(0);  
    });  
  });  
});  
});
```

Listing 9.5 Using the Setup Routine (src/calculator.spec.ts)

As you can see in the source code, the `beforeEach` function is placed in a test suite and executed for each test in that suite. If the current test suite has a child suite—that is, a nested `describe`—then the `beforeEach` function will also be executed for all tests of the inner suite. Like the `beforeEach` function, there is an `afterEach` function that is executed after each test of the suite. If you don't want to execute certain routines before each test, but only once per test suite, you can use the `beforeAll` function. The counterpart to this is the `afterAll` function, which is executed once after all tests of the suite.

9.1.8 Skipping Tests and Running Them Exclusively

During the development process, you may temporarily not want to run certain tests because you know that they may not work due to a rebuild and that the error messages will only make the testing results confusing. You can do that by using the `skip` method. This method is defined for both tests and test suites. For example, to skip the second addition test, you can insert `.skip` between the `it` and the opening parenthesis. This way, you make sure that the test will no longer be executed. The corresponding source code is shown in [Listing 9.6](#):

```
import Calculator from './calculator';

describe('Calculator', () => {
  describe('add', () => {
    let calculator: Calculator;
    beforeEach(() => {
      calculator = new Calculator();
    });

    it('should add 1 and 1 and return 2', () => {
      const result = calculator.add(1, 1);
      expect(result).toBe(2);
    });

    it.skip('should add -1 and 1 and return 0', () => {
      const result = calculator.add(-1, 1);
      expect(result).toBe(0);
    });
  });
});
```

Listing 9.6 Skipping Tests Using the “skip” Method (src/calculator.spec.ts)

When you run the tests, the summary will contain the information that one of the tests was skipped.

```

PASS src/calculator.spec.ts
  Calculator
    add
      ✓ should add 1 and 1 and return 2 (1 ms)
      ○ skipped should add -1 and 1 and return 0

Test Suites: 1 passed, 1 total
Tests:       1 skipped, 1 passed, 2 total
Snapshots:  0 total
Time:        0.574 s, estimated 1 s
Ran all test suites related to changed files.

Watch Usage
> Press a to run all tests.
> Press f to run only failed tests.
> Press q to quit watch mode.
> Press p to filter by a filename regex pattern.
> Press t to filter by a test name regex pattern.
> Press Enter to trigger a test run.

```

Figure 9.2 Summary of the Testing Results

As you can see in [Figure 9.2](#), the number of skipped tests is also highlighted in color. During development, you should keep track of the number of skipped tests and be careful not to skip any tests. The longer a test is disabled, the greater the effort required to run it successfully again. The last resort is to delete and rewrite the test.

As with skipping a single test, you can also insert a `.skip` in `describe` to disable the test suite. You can use the `skip` method as often as you like in your tests to disable tests even multiple times.

However, if you are only interested in the result of one test or test suite within a file, you don't need to disable all other tests. Instead, add `.only` to the test you are interested in, as shown in [Listing 9.7](#):

```

import Calculator from './calculator';

describe('Calculator', () => {
  describe('add', () => {
    let calculator: Calculator;
    beforeEach(() => {
      calculator = new Calculator();
    });

    it('should add 1 and 1 and return 2', () => {
      const result = calculator.add(1, 1);
      expect(result).toBe(2);
    });

    it.only('should add -1 and 1 and return 0', () => {
      const result = calculator.add(-1, 1);

```

```

    expect(result).toBe(0);
  });
});
});

```

Listing 9.7 Exclusive Execution of Single Tests per File (src/calculator.spec.ts)

Again, in the summary of the testing results, you'll see the information that one of the tests was skipped.

9.1.9 Handling Exceptions

In a unit test, you call the function to be tested and check the return of this function. With this approach, the success case can be well covered. However, if you venture into limit regions of the implementation, this method is insufficient.

Let's suppose you pass a number to the `add` method that is larger than the valid range of values. In this case, the addition doesn't work correctly and the `add` method should throw an exception. However, an uncaught exception will inevitably cause the test to fail. But you can resort to the `toThrow` matcher, which is able to handle an exception. For this to work, you need to pass a function object to the `expect` function that will be executed by Jest. The testing framework then takes care of catching and checking the exception. [Listing 9.8](#) contains the source code that checks the exception case of the `add` method:

```

import Calculator from './calculator';

describe('Calculator', () => {
  describe('add', () => {
    let calculator: Calculator;
    beforeEach(() => {
      calculator = new Calculator();
    });

    it('should add 1 and 1 and return 2', () => {...});

    it('should add -1 and 1 and return 0', () => {...});

    it('should throw an error, if a string is provided', () => {
      const testFunction = () => calculator.add( ←
        1, Number.MAX_SAFE_INTEGER + 1);
      expect(testFunction).toThrow();
      expect(testFunction).toThrow(Error);
      expect(testFunction).toThrow('Please provide a valid number');
      expect(testFunction).toThrow(/valid number/);
    });
  });
});

```

```
    });  
  });  
});
```

Listing 9.8 Check for Exceptions (src/calculator.spec.ts)

As you can see in the test code, there are several ways to check a thrown exception. These range from just the fact that an exception of any type has been thrown to type-checking the exception and comparing against the thrown message to comparing the thrown message to a regular expression. The corresponding implementation of the `add` method is shown in [Listing 9.9](#):

```
export default class Calculator {  
  add(a: number, b: number) {  
    if (a > Number.MAX_SAFE_INTEGER || b > Number.MAX_SAFE_INTEGER) {  
      throw new Error('Please provide a valid number');  
    }  
    return a + b;  
  }  
}
```

Listing 9.9 Implementation of the “add” Method (src/calculator.ts)

9.1.10 Testing Asynchronous Operations

The last feature I want to present to you in the context of the basic principles of Jest is handling asynchronous operations. If the result of an operation is available not immediately, but at a later time, the problem arises that the test is finished before the result is available and can be checked. There are several approaches to dealing with such problems.

As a basis for these examples, we use an `Async` class with two different functions. One of them accepts a callback function that is called after one second with a certain value. This method is named `withCallback`. The second function uses the `Promise` API of JavaScript and resolves the `Promise` object after one second with a specified value. The name of this method is `withPromise`.

```
export default class Async {  
  withCallback(callback: (value: string) => void) {  
    setTimeout(() => callback('Hello World'), 1000);  
  }  
  withPromise() {  
    return new Promise(resolve => this.withCallback(resolve));  
  }  
}
```

Listing 9.10 Implementation of the “Async” Class (src/async.ts)

Handling Asynchronous Operations Using the “done” Callback Function

The classic variant for handling asynchronous operations is to use a callback function in the test function. Usually this callback function is named `done`. The `done` function is executed once the asynchronous operation has been completed and the result has been checked.

For Jest, the presence of the `done` parameter means that it is an asynchronous test and the framework must wait for the function to be called. This waiting time is not unlimited but lasts at most until a configured timeout. The default value for this is five seconds. [Listing 9.11](#) shows an example of using the `done` callback function:

```
import Async from './async';

describe('Async', () => {
  let myAsync: Async;
  beforeEach(() => {
    myAsync = new Async();
  });
  it('should work with callback', (done) => {
    myAsync.withCallback((value: string) => {
      expect(value).toBe('Hello World');
      done();
    });
  });
});
```

Listing 9.11 Handling Asynchronous Operations (src/async.spec.ts)

As you can see, the test receives the `done` function as an argument. The callback function passed to the `withCallback` method contains the checking of the value with the `toBe` matcher and the call of the `done` function.

If you now remove the `done` parameter and its call, nothing will happen at first. In the next step, you add `.not` before the `toBe` matcher to provoke a failure of the test, if the test doesn't fail as would be expected. The reason for this is that the test is run immediately, whereas the check takes place after one second. Such modifications, where you intentionally make tests fail, are a proven tool to determine if a test is really running the way you expect it to.

Using Promises

Checking promise-based interfaces is even easier than using callbacks. Once you start using promises, you can use these objects as return values in a test. As a result, the testing framework knows that it must wait for the `Promise` object to resolve. For the test of the `withPromise` method, this means that you call the method as shown in [Listing 9.12](#),

then use the `then` method to express your expectation, and use this entire construct as the return value.

```
import Async from './async';

describe('Async', () => {
  let myAsync: Async;
  beforeEach(() => {
    myAsync = new Async();
  });
  it('should work with callback', () => {...});
  it('should work with promises', () => {
    return myAsync.withPromise().then(value => {
      expect(value).toBe('Hello World');
    });
  });
});
```

Listing 9.12 Testing of Promise-Based Methods (src/async.spec.ts)

Using “resolves” and “rejects”

An alternative to returning the Promise object is to use the `resolves` or `rejects` method. This method must be inserted between `expect` and the matcher and ensures that the Promise object will be handled correctly. Note that you must insert the `return` keyword before the checking starts. This ensures that the test won't be completed before the asynchronous operation has been tested. You can see the corresponding source code in [Listing 9.13](#):

```
import Async from './async';

describe('Async', () => {
  let myAsync: Async;
  beforeEach(() => {
    myAsync = new Async();
  });
  it('should work with callback', () => {...});
  it('should work with promises', () => {...});
  it('should work with resolves', () => {
    const promise = myAsync.withPromise();
    return expect(promise).resolves.toBe('Hello World');
  });
});
```

Listing 9.13 Using the “resolves” Method (src/async.spec.ts)

Implementation as Asynchronous Function

The fourth and last variant to deal with asynchronicity is to implement the test function itself as an `async` function. This allows you to wait for the result of the `Promise` object with the `await` keyword and then perform the check:

```
import Async from './async';

describe('Async', () => {
  let myAsync: Async;
  beforeEach(() => {
    myAsync = new Async();
  });
  it('should work with callback', () => {...});
  it('should work with promises', () => {...});
  it('should work with resolves', () => {...});
  it('should work with async functions', async () => {
    const data = await myAsync.withPromise();
    expect(data).toBe('Hello World');
  });
});
```

Listing 9.14 Checking Asynchronicity with “async” Tests (src/async.spec.ts)

As in the other examples, you can also insert `.not` in [Listing 9.14](#) to intentionally cause the test to fail and ensure that the asynchronous operation is checked correctly.

After this somewhat detailed introduction to the Jest testing framework, we’ll now turn our attention to testing a React application and start with testing helper functions.

9.2 Testing Helper Functions

When developing applications, you should try to swap out as much of your business logic as possible from your components into standalone functions. First, this has the advantage that you develop the logic independently of React, and you can reuse these functions separately from the application. Second, testing such independent functions is much easier as in most cases here you have only inputs through the parameters list and outputs with the return value. Nevertheless, there are some aspects you should consider.

If your function has a side effect in addition to input and output, you must be sure to also check this in the test. If you receive data from a server, you need to make sure that the environment is stable and that your test always returns the same result. You’ll learn more about using sever communication later in this chapter when it comes to checking components with server connections.

In general, it's important that you have your testing environment under control. An environment over which you have no control always presents a difficulty for a test. A classic example of such an uncontrollable environment is one involving random numbers, which you can generate via the `Math.random` method. As an example, we'll use a `getNumber` function that returns a random integer between 0 and 10 (see [Listing 9.15](#)).

To take the randomness out of the game here and ensure that `Math.random` becomes more reliable, Jest provides you with so-called mocks. These are structures that replace parts of the application and that you can control. This makes it possible to specify which value `Math.random` returns in the test, resulting in more stable and predictable tests.

```
function getNumber(): number {  
  return Math.floor(Math.random() * 10);  
}
```

```
export default getNumber;
```

Listing 9.15 Implementation of the “getNumber” Function (`src/getNumber.ts`)

The `Math.random` method returns a floating point number between 0 and 1. You need to multiply this by 10 and then truncate everything after the decimal point using the `Math.floor` method.

[Listing 9.16](#) contains the test for the `getNumber` function:

```
import getNumber from './getNumber';  
  
describe('getNumber', () => {  
  it('should return a valid number', () => {  
    const originalRandom = global.Math.random;  
    global.Math.random = jest.fn().mockReturnValue(0.41);  
  
    const result = getNumber();  
  
    expect(result).toBe(4);  
    expect(global.Math.random).toHaveBeenCalled();  
    global.Math.random = originalRandom;  
  });  
});
```

Listing 9.16 Test for the “getNumber” Function (`src/getNumber.spec.ts`)

As you can see in [Listing 9.16](#), the global `Math.random` method is replaced by a mock function. You can get this by calling `jest.fn`. This method returns an empty mock function. Finally, the `mockReturnValue` method sets the return value of the function. Each

time the `Math.random` method is called, the value 0.41 is returned. For this reason, the `getNumber` function returns the number 4. For example, if you change the value from 0.41 to 0.21, the value 2 will be returned and the test will fail. This behavior can be reproduced for each test run, and you no longer have a problem with random values. Here, you should also follow the established pattern of leaving the environment clean, and you should restore the `Math.random` method.

In addition to checking the return value, you can see in the code example that you have the ability to check various aspects of the call of a mock function—in this case, whether the function was called during the test. You can also check the arguments and return values.

The procedures presented so far are very well suited for testing functions where you have a defined input and an expected return value. When checking components, you also use the basic features of Jest, but you have to adjust the approach a bit.

9.3 Snapshot Testing

The simplest way to test a component is to carry out *snapshot testing*. The snapshots used here are structural images of the components. This feature is not specific to React, but is part of Jest and can be used for other frameworks as well. Snapshot tests are pretty much UI-oriented and are only used to ensure that the structure of the component hasn't changed unintentionally as a result of a change to the code.

The basic snapshot testing procedure is as follows:

1. A snapshot is created during the first test run.
2. When the test is run again, the result is compared with the existing snapshot.
3. If both structures match, the test is considered successful.
4. If there is a difference, you'll receive a corresponding error message.
5. In case of a failure, you have the option to create a new snapshot.

Because snapshot testing is a feature of Jest, the snapshot matcher has no knowledge of React components, so you have to address the component serialization yourself. This is where `react-test-renderer` comes into play.

You can use the `npm install --save-dev react-test-renderer @types/react-test-renderer` command to install the additional package and its type definitions. The test renderer takes care of rendering components into JavaScript structures for the testing environment. The test renderer allows you to render and check components independently of the actual environment (such as the DOM). The `BooksList` component from [Listing 9.17](#) is a typical example of a React component that you can secure with such a snapshot test:

```
import React from 'react';
import { Book } from './Book';
import { StarBorder, Star } from '@mui/icons-material';

type Props = {
  book: Book;
  onRate: (bookId: number, rating: number) => void;
};

const BooksListItem: React.FC<Props> = ({ book, onRate }) => {
  return (
    <tr>
      <td>{book.title}</td>
      <td>{book.author}</td>
      <td>{book.isbn}</td>
      <td>
        {Array(5)
          .fill('')
          .map((value, index) => (
            <button key={index} onClick={() => onRate(↵
              book.id, index + 1)}>
              {book.rating < index + 1 ? <StarBorder /> : <Star />}
            </button>
          ))}
      </td>
    </tr>
  );
};

export default BooksListItem;
```

Listing 9.17 Source Code of the “BooksListItem” Component (src/BooksListItem.tsx)

The `BooksListItem` component is a manageable component that is well suited for a first snapshot test. For the test, you first create a new file named *BooksListItem.spec.tsx*. As in the previous examples, this file contains the test suites and the actual tests. [Listing 9.18](#) shows the source code of the snapshot test:

```
import renderer from 'react-test-renderer';
import BooksListItem from './BooksListItem';

describe('BooksListItem', () => {
  describe('Snapshots', () => {
    it('should match the snapshot', () => {
      const book = {
```

```

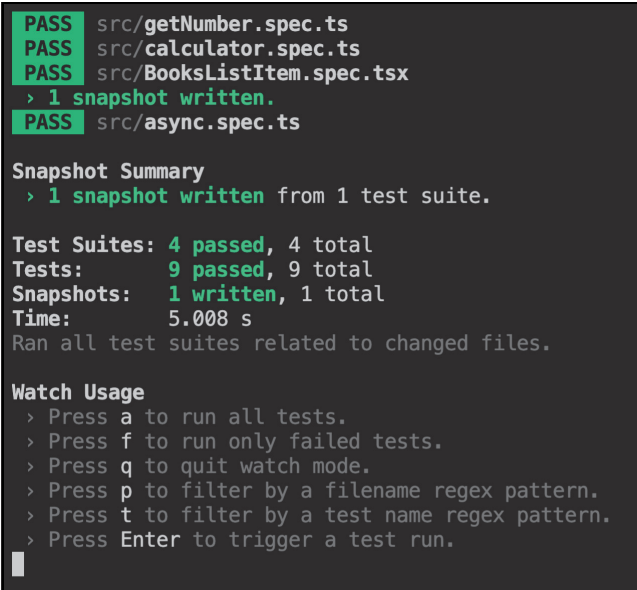
    id: 2,
    title: 'Clean Code',
    author: 'Robert Martin',
    isbn: '978-0132350884',
    rating: 2
  });

  const snapshot = renderer
    .create(<BooksListItem book={book} onRate={() => {}} />)
    .toJSON();
  expect(snapshot).toMatchSnapshot();
});
});
});

```

Listing 9.18 Snapshot Test for the “BooksListItem” Component (src/BooksListItem.spec.ts)

You can use this structure to run the `npm test` command in your application. As you can see in [Figure 9.3](#), the familiar output of the command has changed. Jest informs you here that a snapshot has been created.



```

PASS src/getNumber.spec.ts
PASS src/calculator.spec.ts
PASS src/BooksListItem.spec.tsx
  > 1 snapshot written.
PASS src/async.spec.ts

Snapshot Summary
  > 1 snapshot written from 1 test suite.

Test Suites: 4 passed, 4 total
Tests:       9 passed, 9 total
Snapshots:  1 written, 1 total
Time:       5.008 s
Ran all test suites related to changed files.

Watch Usage
  > Press a to run all tests.
  > Press f to run only failed tests.
  > Press q to quit watch mode.
  > Press p to filter by a filename regex pattern.
  > Press t to filter by a test name regex pattern.
  > Press Enter to trigger a test run.

```

Figure 9.3 Output of the Test Run with Snapshot

On the first run, the test doesn’t yet have any added value as the reference snapshot must first be created. This snapshot is created as a file named *BooksListItem.spec.tsx.snap* in the `__snapshots__` directory inside the `src` directory. [Listing 9.19](#) contains an extract from this file.

```
// Jest Snapshot v1, https://goo.gl/fbAQLP

exports[`BooksListItem Snapshots should match the snapshot 1`] = `
<tr>
  <td>
    Clean Code
  </td>
  <td>
    Robert Martin
  </td>
  <td>
    978-0132350884
  </td>
  <td>
    <button
      onClick={[Function]}
    >
      <svg
        aria-hidden={true}
        className="MuiSvgIcon-root MuiSvgIcon-fontSizeMedium ↵
          css-i4bv87-MuiSvgIcon-root"
        data-testid="StarIcon"
        focusable="false"
        viewBox="0 0 24 24"
      >
        <path
          d="M12 17.27 18.18 21l-1.64-7.03L22 9.24l-7.19-.61L12 2 9.19
            8.63 2 9.24l5.46 4.73L5.82 21z"
        />
      </svg>
    </button>
    <button onClick={[Function]}>...</button>
    <button onClick={[Function]}>...</button>
    <button onClick={[Function]}>...</button>
    <button onClick={[Function]}>...</button>
  </td>
</tr>
`;
```

Listing 9.19 Structure of a Jest Snapshot (src/__snapshots__/Card.spec.tsx.snap)

If you now change the structure of the component or the data of the test, the test will fail, as shown in [Figure 9.4](#).

```

FAIL src/BooksListItem.spec.tsx
  ● BooksListItem > Snapshots > should match the snapshot

    expect(received).toMatchSnapshot()

    Snapshot name: `BooksListItem Snapshots should match the snapshot 1`

    - Snapshot - 1
    + Received + 1

    @@ -1,11 +1,11 @@
      <tr>
        <td>
          Clean Code
        </td>
        <td>
    -   Robert Martin
    +   Robert C. Martin
      </td>
        <td>
          978-0132350884
        </td>
        <td>

    16 |         .create(<BooksListItem book={book} onRate={() => {}} />)
    17 |         .toJSON();
    > 18 |         expect(snapshot).toMatchSnapshot();
       |                             ^
    19 |       });
    20 |     });
    21 |   });

    at Object.<anonymous> (src/BooksListItem.spec.tsx:18:24)

    > 1 snapshot failed.
PASS src/async.spec.ts

```

Figure 9.4 Output in Case of a Failed Snapshot

For the output in [Figure 9.4](#), we changed the author property in the book object from Robert Martin to Robert C. Martin. Jest detects the difference between the reference snapshot and the currently created one and generates a diff output in which the differences are clearly visible. At this point, you have the option of either adjusting your code if the change is an error, or using the `[u]` command to update the failing snapshots. You can use the `[i]` command to refresh snapshots interactively if multiple snapshot tests fail and you want to check them individually. In this case, you can update the single snapshot via `[u]` or skip it by using `[s]`. The `[q]` command terminates the interactive mode.

As an alternative to updating the snapshots in the Jest console, you can update the snapshots using the `--updateSnapshot` command line option (or `-u` in its short form). By using `--testNamePattern`, the update can be restricted to specific tests.

For the snapshot tests to work on all systems, it's important to check the snapshot files into the repository and treat them as regular code of the application.

Besides the `toMatchSnapshot` matcher, there is the `toMatchInlineSnapshot` matcher, which allows you to manage the snapshot directly in the source code of the test and not in a separate file. This task can be done by Jest itself, or you can install the `prettier` package using the `npm install --save-dev prettier` command. In the latter case, `prettier` creates the inline snapshot. On the first run, you execute the `toMatchInlineSnapshot` matcher in the same way as the `toMatchSnapshot` matcher with no arguments. Jest then creates the snapshot and inserts it in the form of a string as an argument directly in the test code. Especially for more extensive components, this variant makes the source code quite unreadable. Therefore, you should switch to the variant with the snapshots in a separate file here.

9.4 Testing Components

When testing a component, you basically proceed in the same way as for an ordinary unit test: you prepare the environment, you perform an operation, and check the result. As with testing a function, you should try to think in terms of inputs and outputs rather than checking the internal structures of a component. With a function component like the `BooksListItem` component, which is responsible for displaying a single item in a list, this isn't possible anyway because you have no access to internal structures of the component. The values you pass to a component are represented by the props. The actions can range from successful initialization to interaction with the rendered structure. Finally, the outputs consist of the rendered structure or a called callback function that you previously passed as a prop.

For class components, it's also possible to access the internal structures in the form of the state. However, you should avoid this too. A better idea is to test the effects of a state change that are visible to the users.

9.4.1 Testing the “BooksListItem” Component

If you've set up your application using Create React App, the tool will already have installed the *React Testing Library* for you. This library makes component testing easier by providing you with numerous helper functions. However, before you start the actual test, you need to make some minor adjustments to the component. As mentioned earlier, when you unit test a component, you test the visual effects of an action. When you render a component, you typically check to see if certain elements are present. To make it easier to locate them and make your tests more robust, you should add `data-testid` properties for the elements you want to locate. [Listing 9.20](#) shows the updated source code of the component:

```
import React from 'react';
import { Book } from './Book';
import { StarBorder, Star } from '@mui/icons-material';
```

```

type Props = {
  book: Book;
  onRate: (bookId: number, rating: number) => void;
};

const BooksListItem: React.FC<Props> = ({ book, onRate }) => {
  return (
    <tr>
      <td data-testid="title">{book.title}</td>
      <td data-testid="author">{book.author}</td>
      <td data-testid="isbn">{book.isbn}</td>
      <td>
        {Array(5)
          .fill('')
          .map((value, index) => (
            <button
              key={index}
              onClick={() => onRate(book.id, index + 1)}
              data-testid="rating"
            >
              {book.rating < index + 1 ? (
                <StarBorder data-testid="notRated" />
              ) : (
                <Star data-testid="rated" />
              )}
            </button>
          ))}
      </td>
    </tr>
  );
};

export default BooksListItem;

```

Listing 9.20 Adding “data-testid” Properties to the “BooksListItem” Component (src/BooksListItem.tsx)

In the component, you add the `data-testid` property for each of the table cells that contain the title, author, and ISBN. You also add this property to the buttons and star icons that you can use for rating purposes. Having made these preparations, you can now add a new test suite for rendering to the *BooksListItem.spec.tsx* file. [Listing 9.21](#) shows what the associated code looks like:

```
import renderer from 'react-test-renderer';
import { render, screen } from '@testing-library/react';
import BookListItem from './BookListItem';

describe('BookListItem', () => {
  describe('Snapshots', () => {...});
  describe('Rendering', () => {
    it('should render correctly for a given dataset', () => {
      const book = {
        id: 2,
        title: 'Clean Code',
        author: 'Robert C. Martin',
        isbn: '978-0132350884',
        rating: 4,
      };
      render(
        <table>
          <tbody>
            <BookListItem book={book} onRate={() => {}} />
          </tbody>
        </table>
      );
      expect(screen.getByTestId('title')).
        toHaveTextContent('Clean Code');
      expect(screen.getByTestId('author')).toHaveTextContent(
        'Robert C. Martin'
      );
      expect(screen.getByTestId('isbn')).toHaveTextContent(
        '978-0132350884'
      );
      expect(screen.getAllByTestId('rating')).toHaveLength(5);
      expect(screen.getAllByTestId('rated')).toHaveLength(4);
      expect(screen.getAllByTestId('notRated')).toHaveLength(1);
    });
  });
});
```

Listing 9.21 Test of the Rendering of the “BookListItem” Component
(src/BookListItem.spec.tsx)

In the test, you first prepare the data record that you want to display by using the component. Then, you use the `render` function of the React Testing Library to render the component. At that point, you want to make sure that the `BookListItem` component returns a `tr` element as the root element. If you render the component directly, you get

a warning that a `tr` element must not occur in a `div` element. That's because Jest represents the component structure in a `div` element. To work around this problem, you render the component in a combination of `table` and `tbody` elements.

After these two steps, the arrange step and the act step, it's time to review the result. You can use the `screen` object to perform these checks. Alternatively, the `render` function also returns an object that contains, for example, the `getByTestId` method.

The `getByTestId` method returns a reference to the first element that has the specified `data-testid` property. With this reference, you can then use the `toHaveTextContent` matcher to check if, for example, the title is displayed correctly.

Using the `getAllByTestId` method, you get all elements with the respective `data-testid` property. This allows you to check, for example, whether the presentation of the rating is correct.

When you run your tests via the `npm test` command, you should get a success message that all tests have been run successfully.

9.4.2 Testing the Interaction

The `BooksListItem` component is not only used to display a data record, but also has an interface that allows users to interact with the component. The five `button` elements can be used to evaluate the data record. Clicking on one of the `button` elements triggers the `onRate` function, which is passed to the component as a `prop`. You can also secure this aspect of a component by means of a unit test. For this purpose, you render your component, execute the action (i.e., the click), and then check whether a corresponding response (i.e., the call of the `onRate` function) has occurred. [Listing 9.22](#) contains the source code of the test:

```
import renderer from 'react-test-renderer';
import { fireEvent, render, screen } from '@testing-library/react';
import BooksListItem from './BooksListItem';

describe('BooksListItem', () => {
  describe('Snapshots', () => {...});
  describe('Rendering', () => {...});
  describe('Rating', () => {
    it('should call the onRate function correctly', () => {
      const book = {
        id: 2,
        title: 'Clean Code',
        author: 'Robert C. Martin',
        isbn: '978-0132350884',
        rating: 4,
      };
    });
  });
});
```

```
const onRate = jest.fn();
render(
  <table>
    <tbody>
      <BooksListItem book={book} onRate={onRate} />
    </tbody>
  </table>
);
fireEvent.click(screen.getAllByTestId('rating')[2]);
expect(onRate).toHaveBeenCalledWith(2, 3);
});
});
});
```

Listing 9.22 Checking Interactions (src/BooksListItem.spec.tsx)

Compared to the previous test, you adapt the arrange step in such a way that you create a spy function in addition to the data record. You have already become familiar with the test doubles of Jest in the context of the mock function to replace the behavior of `Math.random`. Now the spy function that you create via `jest.fn` is simply used to check the interaction with the component. When the button is clicked on, the `onRate` function is indirectly called with certain parameters.

When rendering the component, you must again make sure that the structure is correct and that both the data record to be displayed and the `onRate` function are passed correctly. Then you use the `click` method of the `fireEvent` object to trigger a click event on the third rating button. This causes the `onRate` function to be called with the values 2 for the ID and 3 for the rating.

The “act” Function

If you research the topic of React and unit tests on the internet, sooner or later you’ll come across the `act` function. This feature allows you to encapsulate actions such as rendering a component or interactions such as a click. The `act` function makes your test behave similarly to the actual execution of React in the browser.

If you use the React Testing Library, you no longer have to bother with using the `act` function because all of the library’s helper functions use the `act` function internally, ensuring that your tests and the application behave in a similar way in the browser.

You can use the `toHaveBeenCalledWith` matcher in conjunction with the spy function to check whether the function was called correctly. When you run your tests with this state, you’ll see that all tests are executed successfully.

9.5 Dealing with Server Dependencies

So far, you've only tested one component that receives the data to be displayed as props. However, you often have to deal with a situation where the component itself takes care of loading the data. Such a server connection is an external dependency over which you have only limited control. This has several consequences for your tests. First, you have to make sure that the server is available for your tests and always sends the same data without any additional tools. Also, your tests depend heavily on the availability of the server. If the server doesn't respond or doesn't respond in time, the test will fail. Another negative aspect of this type of server communication is that the runtime of your tests is significantly extended.

You can avoid all these disadvantages by communicating with a mock backend instead of a server. For this purpose, there are numerous libraries available that do much of the work for you. In this chapter, we'll introduce the Mock Service Worker library.

First you need to install the `msw` package via the `npm install --save-dev msw` command. The `BooksList` component you're now going to test represents a simple list of books. It has a state hook and an effect hook. The effect hook makes sure that the data for display is loaded from the server and stored in the state when the component is mounted. The source code of the component is shown in [Listing 9.23](#):

```
import React, { useEffect, useState } from 'react';
import { Book } from './Book';

const BooksList: React.FC = () => {
  const [books, setBooks] = useState<Book[]>([]);
  useEffect(() => {
    fetch('books/')
      .then((response) => response.json())
      .then((data) => setBooks(data));
  }, []);

  return (
    <ul>
      {books.map((book) => (
        <li key={book.id} data-testid="book">
          {book.title}
        </li>
      ))}
    </ul>
  );
};

export default BooksList;
```

Listing 9.23 Source Code of the “BooksList” Component (src/BooksList.tsx)

For the test, the first step is to set up and activate Mock Service Worker. The test itself then remains almost unaffected by the server communication, as you can see in [Listing 9.24](#):

```
import { render, screen } from '@testing-library/react';
import BooksList from './BooksList';
import { rest } from 'msw';
import { setupServer } from 'msw/node';

const books = [...];

const server = setupServer(
  rest.get('/books', (req, res, ctx) => {
    return res(ctx.json(books));
  })
);

describe('BooksList', () => {
  beforeAll(() => server.listen());
  afterEach(() => server.resetHandlers());
  afterAll(() => server.close());
  it('should render initially', async () => {
    render(<BooksList />);

    const books = await screen.findAllByTestId('book');
    expect(books).toHaveLength(3);
    expect(books[0]).toHaveTextContent('JavaScript -
      The Comprehensive Guide');
    expect(books[1]).toHaveTextContent('Clean Code');
    expect(books[2]).toHaveTextContent('Design Patterns');
  });
});
```

Listing 9.24 Test with Server Dependency (src/BooksList.spec.tsx)

In the test, you first define a `books` array with three entries. Then you use the `setupServer` function from the `msw/node` package to create the mock backend. You can model a full-fledged rest interface via the `rest` object. In the code example, you define a request handler for GET requests to the `/books` path and return the `books` array.

Prior to all tests, you need to start the server using the `listen` method. After each test, you reset the request handlers so that each subsequent test runs in a clean environment. Once all tests have been run, you close the server again using the `close` method.

You implement the test function as an `async` function because you have to use the `findAllByTestId` method with promises. In the test function, you first render the

component. If you then access the elements right away via `getAllById`, you'll get an error message. The reason is that React initially renders the component with an empty array and therefore doesn't display any records. Not until the effect hook has been processed, the server communication has finished, and the state has been updated will React display the data. You can use the asynchronous `findAllById` method to wait for the list entries to appear and then perform your checks.

9.5.1 Simulating Errors during Communication

During communication with a server, a wide variety of problems can occur. For example, the server may respond with a 404 status code if the requested resource was not found, or it may report a 500 status code if some other server-side problem occurred. In your application, you should handle such cases and display an appropriate error message to your users. In [Listing 9.25](#), you can see the adjusted source code of the `BooksList` component:

```
import React, { useEffect, useState } from 'react';
import { Book } from './Book';

const BooksList: React.FC = () => {
  const [error, setError] = useState(false);
  const [books, setBooks] = useState<Book[]>([]);
  useEffect(() => {
    (async function () {
      const response = await fetch('books/');
      if (response.ok) {
        const data = await response.json();
        setBooks(data);
      } else {
        setError(true);
      }
    })();
  }, []);

  if (error) {
    return <div data-testid="error">An error ↵
      has occurred!</div>;
  } else {
    return (
      <ul>
        {books.map((book) => (
          <li key={book.id} data-testid="book">
            {book.title}
          </li>
        ))}
      </ul>
    );
  }
}
```



```
    ))}
  </ul>
);
}
};

export default BooksList;
```

Listing 9.25 “BooksList” Component with Error Message (src/BooksList.tsx)

In the `BooksList` component, you add an additional `error` Boolean state. This has the value `false` if there is no error and `true` if the server does report an error. In the effect hook, you implement an `async` function that you call immediately and that encapsulates the server communication. If the `ok` property of the `response` object representing the server's response has the value `false`, you set the error state to the value `true`, and React renders the component again with the error message.

In the test, you make sure that the mock server responds with an error code (in this specific case, status code 500 for an internal server error), and then check that the error message displays correctly. You can see the source code of the respective test in [Listing 9.26](#):

```
import { render, screen } from '@testing-library/react';
import BooksList from './BooksList';
import { rest } from 'msw';
import { setupServer } from 'msw/node';

const books = [...];

describe('BooksList', () => {
  describe('Success', () => {...});
  describe('Error', () => {
    let server;
    beforeAll(() => {
      server = setupServer(
        rest.get('/books', (req, res, ctx) => {
          return res(ctx.status(500), ctx.text(↵
            'Internal Server Error'));
        })
      );
    });
    server.listen();
  });
  afterEach(() => server.resetHandlers());
  afterAll(() => server.close());
  it('should render initially', async () => {
    render(<BooksList />);
```

```

    const error = await screen.findByTestId('error');
    expect(error).toBeInTheDocument();
    expect(error).toHaveTextContent('An error has occurred!');
  });
});
});

```

Listing 9.26 Testing Error Handling in the “BooksList” Component (src/BooksList.spec.tsx)

For the success and error case, you want to create a separate test suite within the `BooksList` `describe` block. In the `beforeAll` function, you set up the mock server and use the `ctx.status` method to make it respond to a GET request to the `/books` path with an error with status code 500. This preparation allows you to have your component rendered again and to wait for the error message to appear using the `findByTestId` method.

9.6 Summary

In this chapter, you’ve seen how you can use Jest to write tests for your application and thus secure it automatically:

- You now know how to install and integrate Jest into your application. You can also run the tests in your application.
- Using the matchers of Jest, you can formulate the conditions for your test.
- Setup and teardown routines allow you to prepare the environment of your test and clean up the environment after the test, respectively.
- Separate matchers are available for handling exceptions.
- Jest natively supports testing asynchronous functions.
- The *snapshot testing* feature of Jest allows you to have a structural image of your components generated and compare it to a reference snapshot.
- Jest allows for the testing of React components. As a tool, you can install the *React Testing Library*, which provides additional functionality for finding elements and interacting with components.
- The *spy* and *mock functions* of Jest enable you to monitor function calls and influence the behavior of functions.
- Libraries like *Mock Service Worker* help you test components that communicate with a server interface.
- Not only does Mock Service Worker allow you to test the success case, it’s also able to simulate various failure scenarios.

In the next chapter, you’ll learn how to implement forms in React.

Chapter 10

Forms in React

Until now, the user interaction with your application has been limited to simple click events related to specific elements. In most cases, however, these limited options are not sufficient. Especially when it comes to creating or modifying data records, you need to use full-featured forms. In a React application, you can integrate all valid HTML form elements and use them for your purposes.

In this chapter, you'll learn about different types of form handling using the uncontrolled and controlled components. You will also learn how to use external libraries to work with forms using the concrete example of React Hook Form.

10.1 Uncontrolled Components

Quick Start

An *uncontrolled component* is not synchronized with the state of a component. You can access the value of this type of form element via a ref:

```
import React, { useRef } from 'react';
import './App.css';

const App: React.FC = () => {
  const inputRef = useRef<HTMLInputElement>(null);
  function handleClick() {
    console.log(inputRef.current!.value);
  }
  return (
    <div>
      <input type="text" ref={inputRef} />
      <button onClick={handleClick}>ok</button>
    </div>
  );
};

export default App;
```

Listing 10.1 Example of an Uncontrolled Component (src/App.tsx)

To use an uncontrolled component, you create a `ref` and link it to the `ref` prop of the form element. Then you can access the value of the form element via the `current.value` property of `Ref`.

The simplest way to deal with forms is *uncontrolled components*. Their name derives from the fact that React does not exert any control over these form elements, but simply addresses them via references.

As a rule, you should rely on *controlled components* when implementing your application, as they ensure that the variables in your component and the values of the input elements are synchronized. You can then access the variables at any time and always get the current value. The situation is different with uncontrolled components, which are used to access the value by reference—just like in old jQuery apps.

10.1.1 Handling References in React

React takes a declarative approach to implementing the interface of an application. So you describe what you want to achieve instead of how you want to achieve it. This is also why it's considered an antipattern if you explicitly select elements within your application and apply operations to them. But this very pattern is the basis of uncontrolled components. *References* in React (or *refs* for short) give you a way to access the DOM elements of your application without using `querySelector`.

In the past, the `createRef` method represented the core of refs in React. With the introduction of the Hooks API, it was replaced by the *ref hook*. The `useRef` function creates an unbound reference to which you can assign a concrete element. This is done via the `ref` attribute of an element. To demonstrate how you can handle refs as well as uncontrolled components in React, we'll implement a simple login form ahead. Because the examples so far have always been of a manageable size, it's sufficient to place the files directly in the `src` directory of the application. But if you implement a more extensive application, you'll come up against limits with this approach, as the overview is quickly lost with many files.

Organizing the Source Code

Depending on how extensive your application is, different forms of structuring the source code are possible. However, this doesn't mean that you have to know at the beginning of your work how extensive your source code will become. You can start with a lightweight approach and implement additional structures later. With this approach, the structure evolves along with the functional scope. The different approaches are as follows:

- **Lightweight approach**

The current state of the sample application represents the most lightweight variant of the structuring. Here, all files are located in the `src` directory of the application. In

this case, the differentiation of the various file types is done based on the file names. You should make sure that it is obvious at first glance what a file is about. For example, you can give your components the extension *.component.tsx* to make it clear that a file contains a component.

■ Grouping according to the type of the elements

One approach that provides somewhat more structure, but is also only partially suitable for large applications, is to group files based on their type. When doing so, you create directories that represent the different types. This could result in the *components*, *hooks*, *models*, and *util* directories for the sample application. The *components* directory contains the individual component files. If you have many components, you can again create subdirectories per component here, which then contain the component itself, the associated test, and the stylesheet. The *hooks* directory contains the hooks of the application. The *models* directory contains the data classes, and finally the *utils* directory contains files with helper functions.

■ Functional grouping

In a functional grouping approach, the directory structure represents the individual function-based modules of the application. For example, if you have a module in your application that handles user administration, a directory named *user* would be suitable for this purpose. You can organize all components related to the user login in a directory named *login*. The advantage of functional grouping is that this variant can also be used for extensive applications and ensures that the directory structure remains clear even with many components.

Listing 10.2 shows the implementation of the form as a function component:

```
import React, { useRef } from 'react';

const Login: React.FC = () => {
  const usernameRef = useRef<HTMLInputElement>(null);
  const passwordRef = useRef<HTMLInputElement>(null);

  return (
    <form>
      <div>
        <label>
          Username:
          <input type="text" ref={usernameRef} />
        </label>
      </div>
      <div>
        <label>
          Password:
          <input type="password" ref={passwordRef} />
        </label>
      </div>
    </form>
  );
}
```

```
        </label>
      </div>
      <button type="submit">submit</button>
    </form>
  );
};
```

```
export default Login;
```

Listing 10.2 Basic Form with Uncontrolled Components (src/Login.tsx)

The basis of the login form is the JSX structure in the component function. This is a simple form with a text field for the user name and a password field. Both elements are described by means of a label. To submit the form, you use a submit button. The only part specific to React in this form is the `ref` attributes of the two input fields. Using these fields, you can access the form fields from the methods of the component. You can create both references using the `useRef` function of React.

If you use TypeScript, as in the example, `useRef` is implemented as a generic function where you also specify what kind of element the reference points to. In the case of the example, this is the `HTMLInputElement` type. At this point, you can already integrate the `Login` component into the `App` component for a first test. [Listing 10.3](#) shows the source code of the `App` component:

```
import React from 'react';
import './App.css';
import Login from './Login';

const App: React.FC = () => {
  return <Login />;
};
```

```
export default App;
```

Listing 10.3 Integration of the “Login” Component (src/App.tsx)

Autofocus of an Input Field with Refs

Refs can be used to implement features, such as autofocus for a field. Although there is the `autofocus` attribute of an HTML element for this purpose, this attribute isn’t always reliable due to the dynamic nature of React. A better alternative is to use refs for this purpose. To do this, you need to define an effect hook that doesn’t become active until the component is mounted and then execute the `focus` method of the `username` element there. This method is available to you within the `current` property of the Ref object. [Listing 10.4](#) contains the adaptation of the `Login` component:

```
import React, { useEffect, useRef } from 'react';

const Login: React.FC = () => {
  const usernameRef = useRef<HTMLInputElement>(null);
  const passwordRef = useRef<HTMLInputElement>(null);

  useEffect(() => {
    usernameRef.current!.focus();
  }, []);

  return (
    <form>
      <div>
        <label>
          Username:
          <input type="text" ref={usernameRef} />
        </label>
      </div>
      <div>
        <label>
          Password:
          <input type="password" ref={passwordRef} />
        </label>
      </div>
      <button type="submit">submit</button>
    </form>
  );
};

export default Login;
```

Listing 10.4 Autofocus with Refs (src/login/Login.tsx)

By using `!` after the `current` property, you tell the TypeScript compiler that the `current` property is definitely an object, even though it's possible that it could theoretically be `null`. When you open your application in the browser with these changes, the input field for the user name is activated by default and you can start typing there directly.

To complete the login form, you still need to implement a routine that will submit the form.

Submitting the Form

Up to this point, you have only used click events. React provides a few more event handlers in the context of forms. In the case of the login form, for example, you need to

respond to the `submit` event, which you can trigger either by pressing the `Enter` key or by clicking the **Submit** button. Inside the `handleSubmit` method, you implement the logic to respond appropriately to the `submit` event:

```
import React, { FormEvent, useEffect, useRef } from 'react';

type Props = {
  onLogin: (username: string, password: string) => void;
};

const Login: React.FC<Props> = ({ onLogin }) => {
  const usernameRef = useRef<HTMLInputElement>(null);
  const passwordRef = useRef<HTMLInputElement>(null);

  useEffect(() => {
    usernameRef.current!.focus();
  }, []);

  function handleSubmit(event: FormEvent<HTMLFormElement>) {
    event.preventDefault();
    onLogin(usernameRef.current!.value, passwordRef.current!.value);
  }

  return (
    <form onSubmit={handleSubmit}>
      <div>
        <label>
          Username:
          <input type="text" ref={usernameRef} />
        </label>
      </div>
      <div>
        <label>
          Password:
          <input type="password" ref={passwordRef} />
        </label>
      </div>
      <button type="submit">submit</button>
    </form>
  );
};

export default Login;
```

Listing 10.5 Handling of the “submit” Event in the “Login” Component (src/Login.tsx)

As you can see in [Listing 10.5](#), the `handleSubmit` function is automatically passed the object representation of the `submit` event. This object is of the `FormEvent` type. The default behavior of an HTML form is to reload the page when the form is submitted. You can prevent this by using the `preventDefault` method of the event object. You access the individual values of the form fields via the two `ref` objects. The exclamation mark after `current` indicates that this value is not `null`, so there will be no TypeScript error. Because a component is responsible for only one aspect, the `Login` component handles the display of the login form and the associated event handling, but not the sending of the login credentials to the server. This task is performed by the `onLogin` function, which is passed into the component via the props.

If you now pass the `onLogin` prop correctly inside the `App` component, you can already test your component. In a minimal implementation, a callback function that accepts the user name and password and prints both pieces of information to the console is sufficient. That brings us to the next topic. At this point, you may also want to write a unit test for your component. In this test, you check if the form submission works after entering the user name and password and if the `onLogin` function is called correctly. For this test, you create a new file named `Login.spec.tsx` in the `src` directory. The test can then look like the one shown in [Listing 10.6](#):

```
import { fireEvent, render, screen } from '@testing-library/react';
import Login from './Login';

describe('Login', () => {
  it('should call onLogin correctly after submitting the form', () => {
    const onLogin = jest.fn();
    render(<Login onLogin={onLogin} />);

    const username = screen.getByTestId('username');
    const password = screen.getByTestId('password');
    const submit = screen.getByTestId('submit');

    fireEvent.change(username, { target: { value: 'testuser' } });
    fireEvent.change(password, { target: { value: 'testpassword' } });
    fireEvent.click(submit);

    expect(onLogin).toHaveBeenCalledWith('testuser', 'testpassword');
  });
});
```

Listing 10.6 Testing the “Login” Component (`src/Login.spec.tsx`)

To make sure your test runs properly, you need to add the `data-test id` attributes to the two input elements and the **Submit** button in the `Login` component. [Listing 10.7](#) shows the adapted JSX code of the `Login` component:

```
<form onSubmit={handleSubmit}>
  <div>
    <label>
      Username:
      <input type="text" ref={usernameRef} data-testid="username" />
    </label>
  </div>
  <div>
    <label>
      Password:
      <input type="password" ref={passwordRef} data-testid="password" />
    </label>
  </div>
  <button type="submit" data-testid="submit">
    login
  </button>
</form>
```

Listing 10.7 Adjustments to the JSX Structure of the “Login” Component (src/Login.tsx)

In the test, you create a spy function for the `onLogin` prop that you can use at the end of the test to verify that the form is working properly. Then render the form, passing the spy function via the `onLogin` prop. In the next step, you use the `getByTestId` method to get access to the individual elements, set the values for the user name and password via `fireEvent.change`, and finally let the test click on the **Submit** button. After that, you use the `toHaveBeenCalledWith` matcher of Jest to check if the spy function was called with the correct user name–password combination.

Thus, while Jest performs the test successfully, your current implementation only covers the success case. If an error occurs, the users don’t learn about it directly.

Error Handling in Uncontrolled Components

However, you don’t have to do without error handling for such form implementations. Depending on the application, this can be done during input, before the form is submitted, or only on the server side. The question which variant you choose depends on the requirements of your application. The sooner you check the input, the faster your user will get a response. In some cases, however, client-side checking is also not possible because the client doesn’t have the required information; then the checking must be done on the server side, and the feedback is somewhat delayed.

In the first variant, you implement an event handler for the `change` event of a form field and check for each change to see if it is a valid value. If that isn’t the case, you can provide direct feedback to the users. However, controlled components are much more suitable for such an implementation as they are directly connected to the component structure. We’ll return to that later.

If you perform the check only when the form is submitted, the feedback to the users is no longer provided immediately upon the input, but before a potentially time-consuming server request. In this case, you implement the check in the `submit` handler and, depending on the result, you either display an error message or submit the data to the server.

In the third variant, the check is performed on the server side. That's required during the login process because the client side doesn't know if the entered credentials are valid.

Warning!

In addition to a client-side check, you should always check user input on the server side as well as the client-side validation can be bypassed by potential attackers.

In the example, you now check if both a user name and a password have been specified. You also need to be able to deal with a failure of the application. To visualize errors, you implement an `error` state in the `Login` component. This state shows an empty string in case of success; only if an error occurs will the corresponding error message be set. In addition to this, the parent component can set an `error` prop to indicate that the login failed. You can see the adjusted version of the `Login` component in [Listing 10.8](#):

```
import React, { FormEvent, useEffect, useRef, useState } from 'react';

type Props = {
  onLogin: (username: string, password: string) => void;
  loginError?: string;
};

const Login: React.FC<Props> = ({ onLogin, loginError }) => {
  const [validationError, setValidationError] = useState<string>('');

  const usernameRef = useRef<HTMLInputElement>(null);
  const passwordRef = useRef<HTMLInputElement>(null);

  useEffect(() => {
    usernameRef.current!.focus();
  }, []);

  function handleSubmit(event: FormEvent<HTMLFormElement>) {
    event.preventDefault();
    let error = 'Please enter a username ←
    and a password.';
    const username = usernameRef.current!.value;
    const password = passwordRef.current!.value;
```

```
    if (username && password) {
      error = '';
      onLogin(username, password);
    }
    setValidationError(error);
  }

  return (
    <form onSubmit={handleSubmit}>
      {loginError && <div data-testid="loginError">{loginError}</div>}
      {validationError && (
        <div data-testid="validationError">{validationError}</div>
      )}
    </div>
    <div>
      <label>
        Username:
        <input type="text" ref={usernameRef} data-testid="username" />
      </label>
    </div>
    <div>
      <label>
        Password:
        <input type="password" ref={passwordRef} data-testid="password" />
      </label>
    </div>
    <button type="submit" data-testid="submit">
      login
    </button>
  </form>
);
};

export default Login;
```

Listing 10.8 Display of Error Messages (src/Login.tsx)

To process the errors, you first add an optional `loginError` to the props type. Then you define a local `validationError` state within the component. The form validation is done within the `handleSubmit` function as mentioned. There you define an error message and check if there is a user name and password. If that's the case, you reset the error message to an empty string and call the `onLogin` function. In the last step, you set the `validationError` state to the currently valid value—that is, either an empty string if successful or the error message if the validation failed. Both error cases lead to the output of a corresponding message in the component. To make sure you can secure both cases with unit tests, you assign a `data-testid` property to each of the containers.

In your existing test of the `Login` component, you can first make sure that neither of the two error messages are displayed in case of success. After that, you take care of checking the representation of the login error. To do this, you render the component in the test with the `loginError` prop and expect the error message to be displayed. The test for the validation error is a bit more elaborate. For this test, you only fill in the input field for the user name and leave the password empty. After submitting the form, you check if the validation error message displays. The source code of the three tests for the `Login` component is shown in [Listing 10.9](#):

```
import { fireEvent, render, screen } from '@testing-library/react';
import Login from './Login';

describe('Login', () => {
  it('should call onLogin correctly after submitting the form', () => {
    const onLogin = jest.fn();
    render(<Login onLogin={onLogin} />);

    const username = screen.getByTestId('username');
    const password = screen.getByTestId('password');
    const submit = screen.getByTestId('submit');

    fireEvent.change(username, { target: { value: 'testuser' } });
    fireEvent.change(password, { target: { value: 'testpassword' } });
    fireEvent.click(submit);

    expect(onLogin).toHaveBeenCalledWith('testuser', 'testpassword');
    expect(screen.queryByTestId('loginError')).not.toBeInTheDocument();
    expect(screen.queryByTestId('validationError')).not.toBeInTheDocument();
  });

  it('should display an external login error', () => {
    render(<Login onLogin={jest.fn()} loginError= ↵
      "Login failed" />);

    const loginError = screen.getByTestId('loginError');
    expect(loginError).toBeInTheDocument();
    expect(loginError).toHaveTextContent('Login failed');
  });

  it('should display an error if the validation fails', () => {
    const onLogin = jest.fn();
    render(<Login onLogin={onLogin} />);

    const username = screen.getByTestId('username');
```

```
const password = screen.getByTestId('password');
const submit = screen.getByTestId('submit');

fireEvent.change(username, { target: { value: 'testuser' } });
fireEvent.click(submit);

const validationError = screen.queryByTestId('validationError');

expect(onLogin).not.toHaveBeenCalled();
expect(screen.queryByTestId('loginError')).not.toBeInTheDocument();
expect(validationError).toBeInTheDocument();
expect(validationError).toHaveTextContent(
  'Please enter a username and password.'
);
});
});
```

Listing 10.9 Tests for the Error Cases in the “Login” Component (src/Login.spec.tsx)

This version of the login process is just an intermediate step; in [Chapter 14](#), which is about central state management in your application, we’ll integrate the login process more deeply into the application.

In the meantime, however, you should still take care of the appearance of the login form so that the users of your application are not put off by what they see. To do this, you add some more `className` props to the JSX structure of the form, as shown in [Listing 10.10](#):

```
<form onSubmit={handleSubmit} className="Login">
  {loginError && (
    <div data-testid="loginError" className="error">
      {loginError}
    </div>
  )}
  {validationError && (
    <div data-testid="validationError" className="error">
      {validationError}
    </div>
  )}
</div>
<label>
  Username:
  <input type="text" ref={usernameRef} data-testid="username" />
</label>
</div>
```

```

<div>
  <label>
    Password:
    <input type="password" ref={passwordRef} data-testid="password" />
  </label>
</div>
<button type="submit" data-testid="submit">
  login
</button>
</form>

```

Listing 10.10 Adjustments to the Structure of the “Login” Component (src/Login.tsx)

After that, you can create a file named *Login.scss* in the *src* directory that contains the styles for the component in the form of an SCSS stylesheet. For the styling to work, you must ensure that the *sass* package is installed in your application. The source code of the stylesheet is shown in [Listing 10.11](#):

```

.Login {
  width: 400px;
  height: 200px;
  margin: 50px auto;
  border: 1px solid black;
  box-shadow: 0 0 5px black;
  display: flex;
  flex-direction: column;
  justify-content: space-around;
  align-items: center;
  padding-left: 20px;
  label {
    width: 400px;
    position: relative;
    display: inline-block;
  }
  input {
    position: absolute;
    left: 120px;
  }
  .error {
    color: red;
  }
}

```

Listing 10.11 Styling of the Login Form (src/login/Login.scss)

You include this stylesheet in the Login component via the import `'./Login.scss'` statement. With these changes, the initial view of your application should look as shown in [Figure 10.1](#).

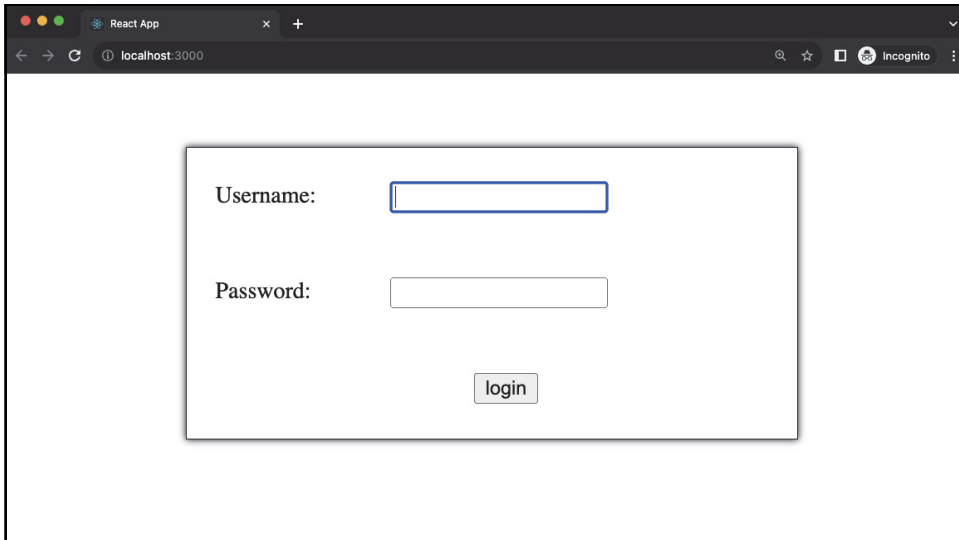


Figure 10.1 Login Form

In addition to the uncontrolled components, there are also controlled components available to implement forms.

10.2 Controlled Components

Quick Start

A *controlled component* is firmly connected with the state of a component. Thus, a change to the form element means a change to the state at the same time and vice versa. For this purpose, you connect the state to the `value` prop of the form element and the `onChange` handler to the setter of the state:

```
import React, { useState } from 'react';
import './App.css';

const App: React.FC = () => {
  const [name, setName] = useState('');

  return (
    <input
      type="text"
      value={name}
```

```

      onChange={(event) => setName(event.target.value)}
    />
  );
}

```

```
export default App;
```

Listing 10.12 Implementation of a Controlled Component (src/App.tsx)

In React applications, you come across controlled components much more frequently than the uncontrolled variant. The main reason is that controlled components are much more comfortable to use. To give you a feel for how to use this type of form implementation, in the following sections we implement a form for managing books, which you can use to create new data records, but also modify existing ones. The idea behind *controlled components* is that a form element manages its own state and you synchronize it with the state of a component. In the first step, you create a new file named *Form.tsx* in the *src* directory. The core of the form implementation is the `Form` component, which you implement as a function component:

```

import React, {
  ChangeEvent,
  FormEvent,
  ReactElement,
  useEffect,
  useState,
} from 'react';
import { Book, InputBook } from './Book';

type Props = {
  onSave: (book: InputBook) => void;
  book?: Book;
};

const initialBook: InputBook = {
  title: '',
  author: '',
  isbn: '',
};

const Form: React.FC<Props> = ({ onSave, book: inputBook }) => {
  const [book, setBook] = useState<InputBook>(initialBook);

  useEffect(() => {
    if (inputBook) {

```

```
      setBook(inputBook);
    }
  }, [inputBook]);

function handleChange(event: ChangeEvent<HTMLInputElement>): void {
  setBook((prevBook) => {
    return { ...prevBook, [event.target.name]: event.target.value };
  });
}

function handleSubmit(event: FormEvent<HTMLFormElement>): void {
  event.preventDefault();
  onSave(book);
}

return (
  <form className="Form" onSubmit={handleSubmit}>
    <div>
      <label htmlFor="title">Title:</label>
      <input
        type="text"
        id="title"
        name="title"
        value={book.title}
        onChange={handleChange}
        data-testid="title"
      />
    </div>
    <div>
      <label htmlFor="author">Author:</label>
      <input
        type="text"
        id="author"
        name="author"
        value={book.author}
        onChange={handleChange}
        data-testid="author"
      />
    </div>
    <div>
      <label htmlFor="isbn">ISBN:</label>
      <input
        type="text"
        id="isbn"

```

```

        name="isbn"
        value={book.isbn}
        onChange={handleChange}
        data-testid="isbn"
      />
    </div>
    <div>
      <button type="submit" data-testid="submit">
        save
      </button>
    </div>
  </form>
);
};

export default Form;

```

Listing 10.13 Structure of the Form Component (src/Form.tsx)

To make sure your form can handle both new and existing data, you must define another type in addition to the `Book` type. This type is named `InputBook` and has an optional `id` and an optional `rating` property. In TypeScript, you can achieve this by using the `Omit` type in the `Book.ts` file as shown in [Listing 10.14](#):

```

export type Book = {
  id: number;
  title: string;
  author: string;
  isbn: string;
  rating: number;
};

export type InputBook = Omit<Book, 'id' | 'rating'> & {
  id?: number;
  rating?: number;
};

```

Listing 10.14 Definition of the “InputBook” Type (src/Book.ts)

With the `Omit` type, you use the `Book` type and remove the `id` and `rating` properties. You use the `&` operator to merge the resulting type with another type that makes both properties optional.

The `Form` component accepts an `onSave` function as a prop, which in turn receives an `InputBook` object. You can pass this function from the parent component to the `Form` component to save the data record. The second prop is a `Book` object. This is an existing

data record that you can modify with the `Form` component. This prop is optional, so you don't have to pass it.

However, the two functions of the form—that is, creating new data records and modifying existing ones—lead to a problem: What information is in the state of the component?

The case of a new data record is relatively simple to implement: you define an initial value for the state in which the three mandatory properties of the `InputBook` type have empty strings as values, and pass this object to the `useState` function when it is called. What is more interesting is the processing of an existing data record that you pass to the component using a prop. This is where an effect hook comes into play. If you define the prop that you rename to `inputBook` in the destructuring statement in the example as a dependency of the effect hook, then you can react to the changes of this prop. React executes this effect hook both initially and whenever the prop gets changed. After that, you only need to check if `inputBook` exists and has a value, and then you can write this value directly to the state.

The JSX structure of the `Form` component consists of a `form` element; three `input` elements for entering the title, author, and ISBN, each with an associated `label` element; and finally a **Submit** button for submitting the form. The peculiarity of this form is that each of the `input` elements is directly connected to the state of the component. The `value` property of the element contains the value of the associated state property; in the case of the title, for example, that's `book.title`. This causes the element to always display the value of the state.

The change handler is also important: if you omit it, you won't be able to change the value of the form element. The change handler is the same function for all three fields, so you can swap it out in the form of the `handleChange` function. The function receives the representation of the change event as an argument. If the browser triggers a change event, you change the state and use the spread operator to create a new object with the properties of the previous state. You also set a dynamic property. The name of this property comes from the `name` attribute of the modified `input` element, and the value is the modified value that you can access through the `value` property. This procedure overwrites the value of the original `State` object, thus updating the state and, consequently, the form.

Finally, the last step consists of implementing the `submit` handler for the form. Again, you don't write this function directly into the JSX structure to avoid enlarging it unnecessarily. The `handleSubmit` function uses a call of the `preventDefault` method to ensure that the browser doesn't attempt to send the form directly to the server and reload the page in the process.

This allows you to include the `Form` component in your `App` component and test the result. In [Listing 10.15](#), you can see how to also simulate editing an existing component:

```
import React from 'react';
import './App.css';
import Form from './Form';

const book = {
  id: 1,
  title: 'JavaScript - The Comprehensive Guide',
  author: 'Philip Ackermann',
  isbn: '978-3836286299',
  rating: 5,
};

const App: React.FC = () => {
  return <Form onSave={({book}) => console.log(book)} book={book} />;
};

export default App;
```

Listing 10.15 Integration of the “Form” Component (src/App.tsx)

If you press the **Submit** button, this implementation will output the data record to be saved to the browser console. The result is shown in [Figure 10.2](#).

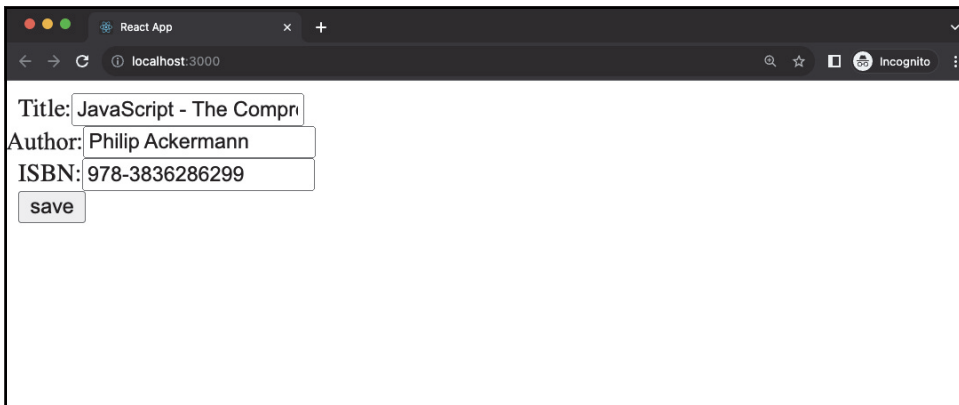


Figure 10.2 Display of the Form in the Browser

If you want to test the creation of a data record, you must remove the `book` prop. This causes React to render the component with the initial data record.

At this point, the `Form` component works; however, if you make a change to the component, you must manually retest the functionality each time. So it's time to secure the component's features by way of unit testing. For this purpose, you create a new file named *Form.spec.tsx* in the *src* directory and implement one test each for creating and modifying data records there. The tests render the component, interact with it, and finally check if the function in the `onSave` prop was executed correctly. The source code of these tests can be found in [Listing 10.16](#):

```
import { fireEvent, render, screen } from '@testing-library/react';
import Form from './Form';

describe('Form', () => {
  it('should create a new Book', () => {
    const onSave = jest.fn();
    render(<Form onSave={onSave} />);
    fireEvent.change(screen.getByTestId('title'), {
      target: { value: 'Design Patterns' },
    });
    fireEvent.change(screen.getByTestId('author'), {
      target: { value: 'Erich Gamma' },
    });
    fireEvent.change(screen.getByTestId('isbn'), {
      target: { value: '978-0201633610' },
    });
    fireEvent.click(screen.getByTestId('submit'));
    expect(onSave).toHaveBeenCalledWith({
      title: 'Design Patterns',
      author: 'Erich Gamma',
      isbn: '978-0201633610',
    });
  });

  it('should modify an existing Book', () => {
    const book = {
      id: 2,
      title: 'Clean Code',
      author: 'Robert Martin',
      isbn: '978-0132350884',
      rating: 2
    };
    const onSave = jest.fn();
    render(<Form onSave={onSave} book={book} />);
    fireEvent.change(screen.getByTestId('author'), {
      target: { value: 'Robert C. Martin' },
    });
    fireEvent.click(screen.getByTestId('submit'));
    expect(onSave).toHaveBeenCalledWith({
      id: 2,
      title: 'Clean Code',
      author: 'Robert C. Martin',
      isbn: '978-0132350884',
      rating: 2
    });
  });
});
```

```

    });
  });
});

```

Listing 10.16 Tests for the “Form” Component (src/Form.spec.tsx)

The first test renders the component without a `book` prop and with a spy function that it passes to the component via the `onSave` prop. After filling in all the fields and submitting the form, the test checks if the spy function was called with the correct object.

The second test ensures that the component is rendered with an existing dataset, also passing a spy function. The test only modifies the author's name and submits the form. It also checks that the modification has been applied and ensures that all other properties of the object have remained unchanged.

The TDD cycle (see [Chapter 9](#)) states that refactorings are allowed in the case of green tests. The `Form` component is relatively extensive and offers potential for improvement. For example, you can use a custom hook to separate the logic from the presentation. To do this, you create a new file named `useForm.ts` in the `src` directory and implement the `useForm` function there:

```

import { useState, useEffect, ChangeEvent, FormEvent } from 'react';
import { InputBook } from './Book';

const initialBook: InputBook = {
  title: '',
  author: '',
  isbn: '',
};

function useForm(
  onSave: (book: InputBook) => void,
  inputBook?: InputBook
): {
  book: InputBook;
  handleChange: (event: ChangeEvent<HTMLInputElement>) => void;
  handleSubmit: (event: FormEvent<HTMLFormElement>) => void;
} {
  const [book, setBook] = useState<InputBook>(initialBook);

  useEffect(() => {
    if (inputBook) {
      setBook(inputBook);
    }
  }, [inputBook]);

```



```
function handleChange(event: ChangeEvent<HTMLInputElement>): void {
  setBook((prevBook) => {
    return { ...prevBook, [event.target.name]: event.target.value };
  });
}

function handleSubmit(event: FormEvent<HTMLFormElement>): void {
  event.preventDefault();
  onSave(book);
}

return {
  book,
  handleChange,
  handleSubmit,
};
}

export default useForm;
```

Listing 10.17 Custom Hook to Swap Out the Logic from the “Form” Component (src/useForm.ts)

To create the `useForm` hook, you cut all the logic from the component and paste it into the new function. In the first step, you must then ensure that all the required structures are imported correctly. Your development environment will usually help you in this regard, creating the required `import` statements for you. Then you define the missing `onSave` and `inputBook` structures as parameters of the function and set the return value, which combines the state—that is, `book`—and the `handleChange` and `handleSubmit` functions into one object.

With this preparation you can integrate the hook function into your `Form` component. You can see how this works in [Listing 10.18](#):

```
import React from 'react';
import { Book, InputBook } from './Book';
import useForm from './useForm';

type Props = {
  onSave: (book: InputBook) => void;
  book?: Book;
};

const Form: React.FC<Props> = ({ onSave, book: inputBook }) => {
  const { book, handleSubmit, handleChange } = useForm(
    onSave, inputBook
  );
```

```

return (
  <form className="Form" onSubmit={handleSubmit}>
    <div>
      <label htmlFor="title">Title:</label>
      <input
        type="text"
        id="title"
        name="title"
        value={book.title}
        onChange={handleChange}
        data-testid="title"
      />
    </div>
    <div>
      <label htmlFor="author">Author:</label>
      <input
        type="text"
        id="author"
        name="author"
        value={book.author}
        onChange={handleChange}
        data-testid="author"
      />
    </div>
    <div>
      <label htmlFor="isbn">ISBN:</label>
      <input
        type="text"
        id="isbn"
        name="isbn"
        value={book.isbn}
        onChange={handleChange}
        data-testid="isbn"
      />
    </div>
    <div>
      <button type="submit" data-testid="submit">
        save
      </button>
    </div>
  </form>
);
};

```

```
export default Form;
```

Listing 10.18 Integration of the “useForm” Hook in the “Form” Component (src/Form.tsx)

Because the `useForm` function provides all the structures you need, you can extract them using a destructuring statement and just need to make sure that you pass the `onSave` function and the optional `inputBook` object to the function correctly.

You can verify that your component still works after the change by manually testing the component in the browser or by running your tests.

Synthetic Events

The handler functions of React don't receive the browser's native event objects, but receive wrapper objects called *synthetic events*. These make sure that events behave consistently across all browsers. Prior to version 17 of React, wrapping events was also done for performance reasons. The events were reused by React and the properties were reset once the event handler was executed. As a consequence, you could no longer directly access the properties of the event objects in operations, such as setting the state with a callback function, but had to store them in variables upfront. Because this optimization did not provide the expected performance boost, but led to a lot of confusion in the community, the React team decided to remove this event pooling.

10.3 File Uploads

So far you've worked with simple text fields. But forms offer numerous other methods of interaction. One of the most challenging is file uploads, which break out of the presented scheme. As an interface to the users, you use an `input` element of type `file` for the upload. You use this element only for reading when submitting the form and therefore set it as an uncontrolled component.

To demonstrate a file upload, you create a form that allows your users to upload an image of a book. To keep the example manageable, you add only a text field for the title next to the file upload. In [Listing 10.19](#), you can find the source code of the `Form` component:

```
import React, {
  ChangeEvent,
  FormEvent,
  useEffect,
  useRef,
  useState,
} from 'react';
import { Book, InputBook } from './Book';

const initialBook: InputBook = {
  title: '',
};
```

```

type Props = {
  onSave: (book: FormData) => void;
  book?: Book;
};

const Form: React.FC<Props> = ({ onSave, book: inputBook }) => {
  const [book, setBook] = useState<InputBook>(initialBook);

  useEffect(() => {
    if (inputBook) {
      setBook(inputBook);
    }
  }, [inputBook]);

  const fileRef = useRef<HTMLInputElement>(null);

  function handleChange(event: ChangeEvent<HTMLInputElement>): void {
    setBook((prevBook) => {
      return { ...prevBook, [event.target.name]: event.target.value };
    });
  }

  function handleSubmit(event: FormEvent<HTMLFormElement>): void {
    event.preventDefault();
    const formData = new FormData();
    formData.append('title', book.title);
    if (fileRef.current?.files && fileRef.current?.files[0]) {
      formData.append('image', fileRef.current?.files[0]);
    }
    onSave(formData);
    setBook(initialBook);
  }

  return (
    <form className="Form" onSubmit={handleSubmit}>
      <div>
        <label htmlFor="title">Title:</label>
        <input
          type="text"
          id="title"
          name="title"
          value={book.title}
          onChange={handleChange}
          data-testid="title"
        />
      </div>
    </form>
  );
}

```

```
    />
  </div>
  <div>
    <label htmlFor="image">Image:</label>
    <input type="file" id="image" ref={fileRef} />
  </div>
</div>
<div>
  <button type="submit" data-testid="submit">
    save
  </button>
</div>
</form>
);
};

export default Form;
```

Listing 10.19 Integration of the File Upload into the Form (src/Form.tsx)

To make the `Form` component work for the simplified structure, you need to modify the `InputBook` data type, remove the `author` and `isbn` properties, and add the optional `image` property instead. The code of the *Book.ts* file is shown in [Listing 10.20](#):

```
export type Book = {
  id: number;
  title: string;
  image?: string;
};

export type InputBook = Omit<Book, 'id'> & {
  id?: number;
};
```

Listing 10.20 Adjusted “InputBook” Data Type for File Upload (src/Book.ts)

But let’s return to the `Form` component: here you adjust the structure of the state so that only the title of the book is managed there. Also, the component no longer returns an `InputBook` object, but a `FormData` type object. This data type is a structure provided by your browser that makes a file upload via a form possible.

In the component itself, you first define a ref for the input element. You adapt the JSX structure and integrate the input element of type `file` and link it to the ref. The final step is to adapt the `handleSubmit` function. Using `preventDefault`, you prevent the browser from automatically submitting the form. Then you create a new `FormData` object and use the `append` method to add first the title and then the file selected for upload. You can access this file via `fileRef.current?.files[0]`.

You pass the `FormData` object, whose type comes from the type declarations for the DOM API, to the `onSave` function that the component received via its props so that the parent component can take care of communicating with the server. The last step in the component consists of resetting the form.

In the parent component—in the example, that’s the `App` component—you take care of the server communication and represent a lightweight variant of the books list. The source code for this is shown in [Listing 10.21](#):

```
import axios from 'axios';
import React, { useEffect, useState } from 'react';
import './App.css';
import { Book } from './Book';
import Form from './Form';

const App: React.FC = () => {
  const [books, setBooks] = useState<Book[]>([]);

  useEffect(() => {
    fetchData();
  }, []);

  async function fetchData() {
    const { data } = await axios.get<Book[]>('http://localhost:3001/books');
    setBooks(data);
  }

  async function handleSubmit(formData: FormData) {
    await axios.post('http://localhost:3001/books', formData, {
      headers: {
        'content-type': 'multipart/form-data',
      },
    });
    fetchData();
  }

  return (
    <div>
      <Form onSave={handleSubmit} />
      <hr />
      <ul>
        {books.map((book) => (
          <li key={book.id}>
            {book.title}
            {book.image && (
```

```
        <img src={book.image} height="40" width="40" alt= ↵
            {book.title} />
      )}
    </li>
  )}
</ul>
</div>
);
};
```

```
export default App;
```

Listing 10.21 Integration of the File Upload into the “App” Component (src/App.tsx)

In the App component, you use the Axios library to send the data to the server. In the `handleSubmit` function, you use the `axios.post` method to create a new data record and then reload the list.

Concerning the display of the list, you assume that the images are uploaded to the *public* directory and that the file name is in the `image` property of the `Book` record. To be able to test the file upload, you need a backend implementation. For this purpose, you first need to create a new subdirectory named *backend* in your application directory. There, you run the `npm init -y` command to create a *package.json* file. After that, you use the `npm install express express-fileupload cors json-server` command to install the required packages and then you can implement your server interface in a new file named *index.js* in the *backend* directory. The corresponding source code is shown in [Listing 10.22](#):

```
import express from 'express';
import jsonServer from 'json-server';
import cors from 'cors';
import fileUpload from 'express-fileupload';
import path from 'path';
import { fileURLToPath } from 'url';

const __filename = fileURLToPath(import.meta.url);
const __dirname = path.dirname(__filename);

const server = express();

server.use(fileUpload());

server.use(express.urlencoded({ extended: true }));
server.use(cors());
```

```

server.use('/books', (req, res, next) => {
  if ((req.method === 'POST' || req.method === 'PUT') && req.files.image) {
    const file = req.files.image;
    const name = req.body.title.replace(' ', '_');
    const image = `${name}.png`;
    file.mv(`${__dirname}/../public/${image}`);
    req.body.image = image;
  } else {
    req.body.image = '';
  }
  next();
});

server.use(jsonServer.router('data.json'));

server.listen(3001);

```

Listing 10.22 Server Interface for the File Upload (backend/index.js)

When receiving a POST or PUT request, the server process takes the submitted file and moves it to the *public* directory of the React application, where you can access it from the frontend. Start your server process via the `node index.js` command in the *backend* directory and run the frontend in parallel using `npm start` in the application directory so that you can create new records with images. [Figure 10.3](#) shows the result in the browser.

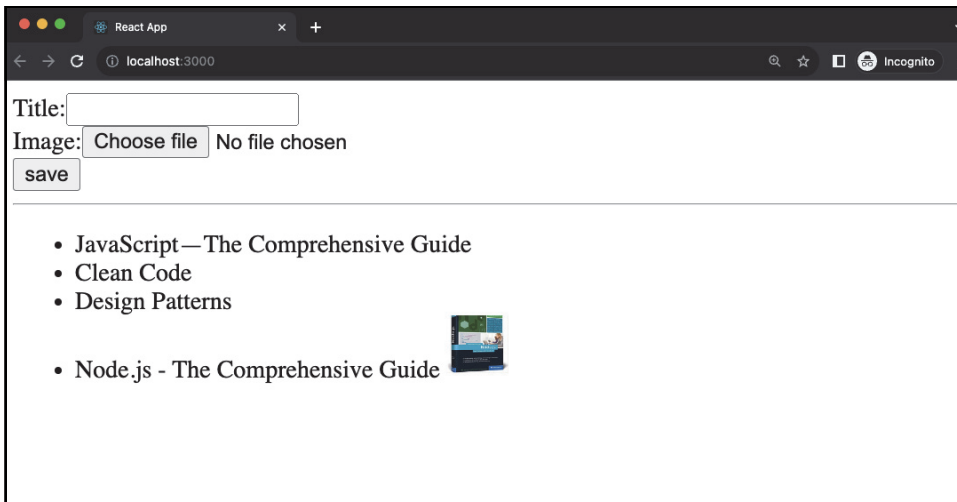


Figure 10.3 Form and List Display with File Upload

The next step in the form implementation is the validation of the input. You can either do this yourself or use an established library. You can insert the validation at various points during the form processing. It's usually most convenient for your users to perform the validation directly during input to give them immediate feedback.

Where you take action here is in the `change` handler, which is executed on every input, giving you the opportunity to react. Because React doesn't provide any functionality for validating form elements, it's up to you to implement it fully. You must also consider borderline cases such as initial empty fields or the like. We'll introduce a library in the following section, React Hook Form, that does a lot of the work for you.

10.4 Form Validation Using React Hook Form

As the name suggests, React Hook Form uses custom hooks and provides you with functionality that allows you to implement React forms in just a few steps and, most importantly, with little additional code. In addition, the library provides you with functionality you need for validation. For example, you can determine whether a user has already made entries in a form and whether those entries are incorrect. You also have access to the respective error messages. But before you can use the React Hook Form library, you must first install it. This is done via the `npm install react-hook-form` command. Because React Hook Form itself is implemented in TypeScript, it also comes with its own type definitions already installed, so you don't have to install them manually.

The core of the library is the `useForm` function. This custom hook provides several properties and functions for handling forms. The most important of these are the `register` function, which enables you to link a form element to React Hook Form, and the `handleSubmit` function, which takes care of submitting the form.

You use the `register` function as a prop for the form elements. It makes sure that the element is connected to the internal state of React Hook Form and represents the respective value. It also takes care of updating the state when the field value is changed by the user, by making React Hook Form respond to the `Change` event. So you no longer have to bother handling the event yourself.

You link the `handleSubmit` function to the `onSubmit` prop of the form element. This ensures that the browser doesn't attempt to send the form to the server, triggering a page load. In [Listing 10.23](#), you can see what the form for managing books from the previous examples looks like with React Hook Form:

```
import React, { useEffect } from 'react';
import { useForm } from 'react-hook-form';
import { Book, InputBook } from './Book';

type Props = {
  onSave: (book: InputBook) => void;
```

```

    book?: Book;
  };

  const defaultValues: InputBook = {
    title: '',
    author: '',
    isbn: '',
  };

  const Form: React.FC<Props> = ({ onSave, book: inputBook }) => {
    const { register, handleSubmit, reset } = useForm<InputBook>({
      defaultValues,
    });

    useEffect(() => {
      reset(inputBook);
    }, [inputBook, reset]);

    return (
      <form className="Form" onSubmit={handleSubmit(onSave)}>
        <div>
          <label htmlFor="title">Title:</label>
          <input type="text" data-testid="title" {...register('title')} />
        </div>
        <div>
          <label htmlFor="author">Author:</label>
          <input type="text" data-testid="author" {...register('author')} />
        </div>
        <div>
          <label htmlFor="isbn">ISBN:</label>
          <input type="text" data-testid="isbn" {...register('isbn')} />
        </div>
        <div>
          <button type="submit" data-testid="submit">
            save
          </button>
        </div>
      </form>
    );
  };

  export default Form;

```

Listing 10.23 Form Implementation with React Hook Form (src/Form.tsx)

At the beginning of the component, you call the `useForm` function of React Hook Form. This function is implemented as a generic function and accepts a type containing the individual form fields and their matching types. You can also pass a configuration object to the function. In the example, you define a set of initial values.

In the function's effect hook, you use the `reset` function of React Hook Form to load data into the form. This is needed if you want to edit a record.

You connect the `onSubmit` prop of the `form` element to the `handleSubmit` function of React Hook Form and pass the `onSave` function to it, which takes care of saving the data and is passed as a prop from the parent component to the `Form` component.

Finally, in the form elements, you call the `register` function with the name of the respective field. The result is an object whose properties you apply to the element using the spread operator.

You can test the component by integrating it in the `App` component. In [Listing 10.24](#), you can see the two use cases supported by the `Form` component: creating a new data record and editing an existing one.

```
import React from 'react';
import './App.css';
import { InputBook } from './Book';
import Form from './Form';

const App: React.FC = () => {
  return (
    <div>
      <Form
        onSave={(book: InputBook) => {
          console.log(book);
        }}
      />
      <hr />
      <Form
        book={{
          id: 1,
          title: 'JavaScript - The Comprehensive Guide',
          author: 'Philip Ackermann',
          isbn: '978-3836286299',
        }}
        onSave={(book: InputBook) => {
          console.log(book);
        }}
      />
    </div>
  );
};
```

```
};
```

```
export default App;
```

Listing 10.24 Integration of the Form Component (src/App.tsx)

Not only does React Hook Form relieve you of busywork, it also helps you to validate your forms.

10.4.1 Form Validation Using React Hook Form

React Hook Form provides you with a lightweight variant of form validation where you define the rules using a configuration object that you pass to the `register` function of each field. You can access the errors of the form via the `formState` property and its `errors` property. You can see the result in the example in [Listing 10.25](#):

```
import React, { useEffect } from 'react';
import { useForm } from 'react-hook-form';
import { Book, InputBook } from './Book';

type Props = {
  onSave: (book: InputBook) => void;
  book?: Book;
};

const defaultValues: InputBook = {
  title: '',
  author: '',
  isbn: '',
};

const Form: React.FC<Props> = ({ onSave, book: inputBook }) => {
  const {
    register,
    handleSubmit,
    reset,
    formState: { errors },
  } = useForm<InputBook>({
    defaultValues,
  });

  useEffect(() => {
    reset(inputBook);
  }, [inputBook, reset]);
```

```
return (  
  <form className="Form" onSubmit={handleSubmit(onSave)}>  
    <div>  
      <label htmlFor="title">Title:</label>  
      <input  
        type="text"  
        data-testid="title"  
        {...register('title', {  
          required: true,  
          minLength: 4,  
          maxLength: 25,  
        })}>  
    </div>  
    {errors.title && errors.title.type === 'required' && (  
      <div>Title is a required field.</div>  
    )}  
    {errors.title && errors.title.type === 'minLength' && (  
      <div>The title must be at least 4 characters long.</div>  
    )}  
    {errors.title && errors.title.type === 'maxLength' && (  
      <div>The title must not exceed 25 characters.</div>  
    )}  
  </div>  
  <div>  
    <label htmlFor="author">Author:</label>  
    <input  
      type="text"  
      data-testid="author"  
      {...register('author', { required: true })}>  
  </div>  
  {errors.author && <div>Author is a required field</div>  
}</div>  
<div>  
  <label htmlFor="isbn">ISBN:</label>  
  <input  
    type="text"  
    data-testid="isbn"  
    {...register('isbn', { required: true })}>  
</div>  
{errors.isbn && <div>ISBN is a required field</div>  
}</div>  
<div>  
  <button type="submit" data-testid="submit">  
    save
```

```

        </button>
      </div>
    </form>
  );
};

```

```
export default Form;
```

Listing 10.25 Form Validation Using React Hook Form (src/Form.tsx)

You define three rules for the `title` field: This is a required field with a minimum length of four characters and a maximum length of 25 characters. The two remaining fields are mandatory fields without any further rules. React Hook Form checks the field values on every input and only for the fields that the user has already enabled. You can use the `formState.errors` property to access the individual errors and display a message depending on the type of error. Once the error has been corrected, the error message disappears.

Another side effect of form validation is that React Hook Form prevents the form from being submitted while errors are active.

The validation using the built-in rules of React Hook Form eventually reaches its limits. In this case, you can define a validation schema and apply it to your form.

10.4.2 Form Validation Using a Schema

A validation schema is an additional data structure that you define outside of the form itself and connect to React Hook Form using an additional library. One of the most popular libraries in this area is Yup. React Hook Form supports other libraries also, such as Zod, Superstruct, and Joi. In the example in [Listing 10.26](#), you can see how you define a schema using Yup and integrate it into your form:

```

import React, { useEffect } from 'react';
import { useForm } from 'react-hook-form';
import { yupResolver } from '@hookform/resolvers/yup';
import * as yup from 'yup';
import { Book, InputBook } from './Book';

```

```

type Props = {
  onSave: (book: InputBook) => void;
  book?: Book;
};

```

```

const defaultValues: InputBook = {
  title: '',
  author: '',

```

```
    isbn: '',
  };

  const schema = yup
    .object({
      title: yup
        .string()
        .min(4, 'The title must be at least 4 characters long.')
        .max(25, 'The title must not exceed 25 characters.')
        .required('Title is a required field.'),
      author: yup.string().required('Author is a required field.'),
      isbn: yup.string().required('ISBN is a required field.'),
    })
    .required();

  const Form: React.FC<Props> = ({ onSave, book: inputBook }) => {
    const {
      register,
      handleSubmit,
      reset,
      formState: { errors },
    } = useForm<InputBook>({
      defaultValues,
      resolver: yupResolver(schema),
    });

    useEffect(() => {
      reset(inputBook);
    }, [inputBook, reset]);

    return (
      <form className="Form" onSubmit={handleSubmit(onSave)}>
        <div>
          <label htmlFor="title">Title:</label>
          <input
            type="text"
            data-testid="title"
            {...register('title')}
            className={errors.title && 'error'}
          />
          {errors.title &&
            <div className="error" data-testid="titleError">
              {errors.title.message}
            </div>
          }
        </div>
      </form>
    );
  };
}
```

```

    </div>
    <div>
      <label htmlFor="author">Author:</label>
      <input
        type="text"
        data-testid="author"
        {...register('author')}
        className={errors.author && 'error'}
      />
      {errors.author &&
        <div className="error" data-testid="authorError">
          {errors.author.message}
        </div>
      }
    </div>
    <div>
      <label htmlFor="isbn">ISBN:</label>
      <input
        type="text"
        data-testid="isbn"
        {...register('isbn')}
        className={errors.isbn && 'error'}
      />
      {errors.isbn && <div className="error" ↵
        data-testid="isbnError">{errors.isbn.message}</div>
      }
    </div>
    <div>
      <button type="submit" data-testid="submit">
        save
      </button>
    </div>
  </form>
);
};

```

```
export default Form;
```

Listing 10.26 Form Validation Using Yup (src/Form.tsx)

For the code to work, you must first install the `@hookform/resolvers` and `yup` packages using the `npm install @hookform/resolvers yup` command.

Yup allows you to define the validation schema for the form as an object structure using the object function. For each property, you can set different rules. Thus, the title is of type string and has a minimum and a maximum length. You can also specify an error message for each rule. You can use these for the display.

The link between Yup and React Hook Form is done in the configuration object that you pass to the `useForm` function. There you use the `resolver` property and set as value the return value of the `yupResolver` function, to which you pass the schema. The validation of the form works as before, with the difference that you can use the error messages previously defined in the schema via the `message` property of the subobjects of the `errors` object.

10.4.3 Styling the Form

For the styling of the form, you create a file named *Form.scss* in the *src* directory. As the file name implies, this is an SCSS stylesheet. For this to work, you must make sure that the `sass` package is installed. You also need to import the styles into the `Form` component via the `import './Form.scss';` line. The source code of the stylesheet is shown in [Listing 10.27](#):

```
.Form {
  div {
    margin: 5px 0;
  }

  label {
    width: 80px;
    display: inline-block;
  }

  input.error {
    border: 2px solid red;
  }

  div.error {
    color: red;
  }
}
```

Listing 10.27 Styling of the “Form” Component (*src/Form.scss*)

With these customizations, you can test your form implementation and create new data records as well as manipulate existing ones. In [Figure 10.4](#), you can see multiple error messages that are displayed when incorrect entries are made.

As soon as you correct an error in the form, the error message as well as the red frame and the corresponding field will be hidden, so you get immediate feedback. The same happens if you turn a valid input into an incorrect one, such as by deleting the author entry.

Title:

The title must be at least 4 characters long.

Author:

ISBN:

ISBN is a required field.

Figure 10.4 Display of Error Messages in the Form

10.4.4 Testing the Form Validation Automatically

Manual tests are useful, but especially with functionalities like a form, a manual test can quickly become laborious if you check all relevant combinations. For this reason, you should implement unit tests for such components. In the following example, we implement two tests for the `Form` component: A first test checks whether the form can be saved with correct information. In the second test, you intentionally include an error that causes the validation to fail so that the form cannot be submitted.

Listing 10.28 contains the source code of the two tests, which you place in the `Form.spec.tsx` file in the `src` directory of your application:

```
import { fireEvent, render, screen, act } from '@testing-library/react';
import Form from './Form';

describe('Form', () => {
  it('should submit the form successfully', async () => {
    const onSave = jest.fn();
    render(<Form onSave={onSave} />);
    fireEvent.change(screen.getByTestId('title'), {
      target: { value: 'Design Patterns' },
    });
    fireEvent.change(screen.getByTestId('author'), {
      target: { value: 'Erich Gamma' },
    });
    fireEvent.change(screen.getByTestId('isbn'), {
      target: { value: '978-0201633610' },
    });
    await act(() => fireEvent.click(screen.getByTestId('submit')));
```

```
    expect(onSave).toHaveBeenCalledWith(
      {
        title: 'Design Patterns',
        author: 'Erich Gamma',
        isbn: '978-0201633610',
      },
      expect.anything()
    );
  });

it('should fail if input is not valid', async () => {
  const onSave = jest.fn();
  render(<Form onSave={onSave} />);
  fireEvent.change(screen.getByTestId('title'), {
    target: { value: 'Design Patterns' },
  });
  await act(() => fireEvent.click(screen.getByTestId('submit')));

  const titleError = screen.queryByTestId('titleError');
  const authorError = screen.getByTestId('authorError');
  const isbnError = screen.queryByTestId('isbnError');

  expect(onSave).not.toHaveBeenCalled();
  expect(titleError).not.toBeInTheDocument();
  expect(authorError).toHaveTextContent('Author is a required field.');
```

In the second test, you make sure that only the two expected error messages are displayed and the third one remains hidden.

```
  expect(isbnError).not.toBeInTheDocument();
});
```

Listing 10.28 Tests for the “Form” Component (src/Form.spec.tsx)

In both tests, you first define a spy function and use it to render the `Form` component. You then interact with the form and simulate user input and submit the form. In the first test, the inputs are valid and the component executes the spy function correctly. In the second test, you make an invalid entry by not specifying the author and ISBN. In the second test, you make sure that only the two expected error messages are displayed and the third one remains hidden.

Another peculiarity of these tests is that you need to wrap the click on the **Submit** button in an asynchronous call of the `act` function of the React Testing Library; otherwise the form handling won’t work properly.

If you run your tests with this state of the source code using the `npm test` command, you’ll get the success message that both tests were run successfully.

10.5 Summary

In this chapter, you learned how to integrate forms into your React application to allow users to create and modify data records:

- *Uncontrolled components* are not linked to the state of a component. This means that you have to take care of synchronizing the values with the state of the component yourself.
- With uncontrolled components, you use refs and can use them to directly access the form elements and implement features such as element autofocus, for example.
- For *controlled components*, you synchronize the state of the component with the form by taking the value displayed in the form element directly from the component's state.
- Changes to controlled components are made via the change handler of the form element.
- Besides standard elements like text fields, checkboxes, or select elements, React also supports file uploads. These elements behave differently from the other elements because they are read-only.
- React makes no assumptions about the validation of forms. You either need to implement the validation yourself or use established solutions such as React Hook Form.
- You can use React Hook Form either as a higher-order component or as a render-props implementation.
- *Yup* allows you to conveniently define validation schemas that you can integrate for validation purposes.
- React Hook Form takes over the manual implementation of controlled components and automatically inserts the required structures.
- For special cases like file uploads, you have to integrate the change handler yourself.

In the next chapter, you'll learn how to include third-party component libraries in your application to simplify styling and add new features, using the Material UI library as an example.

Chapter 11

Component Libraries in a React Application

For web frontends, there are numerous UI libraries. The best-known representatives are Bootstrap and Material Design. For both solutions there are adaptations for React available. At their core, both solutions provide several React components that you can use to implement your frontends. The benefits of such a solution are that the design is coordinated across the component boundaries and thus the interface experiences an improvement in usability. Solutions such as Material in particular provide not only components, but also indications of how to use and combine them. The goal is to provide an ideal user guidance in the application.

The Material UI component library (<https://mui.com>) contains atomic components like buttons and input fields as well as more extensive components like data grids and dialogs. The behavior of the components is usually determined using the passed props, as is common in React. The styling can be customized via CSS. Material UI is based on *Material Design*, a design language developed by Google. At first, predominantly Google products were developed using this method, but the approach has gathered numerous followers in web development. For a comprehensive introduction to Material Design, you should visit <https://material.io>.

In this chapter, you'll use Material UI to provide the sample application with more visual appeal. But before you start beautifying your application, you need to integrate Material UI first.

11.1 Installing and Integrating Material UI

Material UI is available as an NPM package you can install using the `npm install @mui/material @emotion/react @emotion/styled` command. In addition to the actual Material UI package, you also need to install the Emotion styling framework as Material UI is based on it. If you use TypeScript, you don't need to install additional type definitions because Material UI already provides them.

To have a consistent interface, it's important that the font also matches the design. Material UI is optimized for the Roboto font, so you should install and integrate it as well. You can either include this font in the `index.html` file of your application via a *content delivery network* (CDN) using a `link` tag; in this case, however, you're dependent on an existing internet connection as the font file must be downloaded from the CDN. Alternatively, you can install an NPM package containing the font and integrate it directly into your application. This makes you independent of the connection as you

have control over all the components of your application. You can install the package via the `npm install @fontsource/roboto` command. You can import the font by loading it via the module system inside your application. For the standard typography of Material, you should load the 300, 400, 500, and 700 weightings. You can add the required import statements to the *index.tsx* file of your application, as shown in [Listing 11.1](#):

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import App from './App';
import reportWebVitals from './reportWebVitals';

import '@fontsource/roboto/300.css';
import '@fontsource/roboto/400.css';
import '@fontsource/roboto/500.css';
import '@fontsource/roboto/700.css';

const root = ReactDOM.createRoot(
  document.getElementById('root') as HTMLElement
);
root.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
);

reportWebVitals();
```

Listing 11.1 Integration of the Roboto Font (src/index.tsx)

In addition to these packages, there is another optional package named `@mui/icons-material` that you can install if you want to include the Material icons as SVG elements. We’re going to use this variant in the sample application, so you should install the package using the `npm install @mui/icons-material` command.

Once the installation has been completed, you can use the individual components of Material UI. The library has a modular structure, so you can import each component separately. This results in a smaller package size during the build process as unused Material UI components are not included in the package.

11.2 List Display with the “Table” Component

A typical task in web applications is presenting information in a table. For this purpose, you can use the `Table` component of Material UI. In the simplest case, this component represents a table that follows the style guide for Material Design. You can also extend

the table with features like sorting, filtering, or pagination. As a concrete example, you’ll now implement a table view of books.

The base of the table is the `Table` component. Within this component, you use the `TableHead` and `TableBody` components, just as in an ordinary HTML table. The `TableHead` component in turn contains at least one `TableRow` component; each represents an individual row of the table head. Within this component, you insert the `TableCell` component to display an individual cell. The same structure applies to the `TableBody` component. Again, you use the `TableRow` and `TableCell` components for the purpose of structuring. For a better visualization of the number values, you use the `align` prop with the `right` value in the `TableCell` component to right-align the numbers as well as the headings within the cell. The source code of the table display is shown in [Listing 11.2](#). Save this code to the `List.tsx` file in the `src` directory.

```
import React from 'react';
import {
  Paper,
  Table,
  TableBody,
  TableCell,
  TableHead,
  TableRow,
} from '@mui/material';
import { StarBorder, Star } from '@mui/icons-material';
import { Book } from './Book';

type Props = {
  books: Book[];
};

const List: React.FC<Props> = ({ books }) => {
  return (
    <Paper>
      <Table>
        <TableHead>
          <TableRow>
            <TableCell>Title</TableCell>
            <TableCell>Author</TableCell>
            <TableCell>ISBN</TableCell>
            <TableCell>Rating</TableCell>
          </TableRow>
        </TableHead>
        <TableBody>
          {books.map((book) => (
            <TableRow key={book.id}>
```



```
      <TableCell>{book.title}</TableCell>
      <TableCell>{book.author}</TableCell>
      <TableCell>{book.isbn}</TableCell>
      <TableCell>
        {Array(5)
          .fill('')
          .map((rating, index) =>
            book.rating <= index ? (
              <StarBorder key={index} />
            ) : (
              <Star key={index} />
            )
          )}
      </TableCell>
    </TableRow>
  )}}
</TableBody>
</Table>
</Paper>
);
};
```

```
export default List;
```

Listing 11.2 Table Display with Material UI (src/List.tsx)

The information the component is supposed to display is passed to it in the form of a prop named `book` so that the loading of the data is decoupled from the display. The table head consists of one row containing cells for the Title, Author, ISBN, and Rating properties. For rendering the individual rows, you iterate over the passed `books` array. You provide the `TableRow` component with a `key` prop and the `id` value of the current record. Then you create `TableCell` components for the individual properties. For the display of the rating, you use the `StarBorder` and `Star` icons from the `@mui/icons-material` package and display filled or empty stars, depending on the rating.

The `List` component is responsible only for the display. For this reason, the next thing you need to do is implement a component that loads the data from the server and integrates the `List` component. In the simplest case, you can achieve this in the `App` component. The loading of the data is handled by a combination of state and effect hooks using the `Fetch` API of the browser. The corresponding implementation is shown in [Listing 11.3](#):

```
import React, { useEffect, useState } from 'react';
import './App.css';
import { Book } from './Book';
```

```
import List from './List';

const App: React.FC = () => {
  const [books, setBooks] = useState<Book[]>([]);

  useEffect(() => {
    fetch('http://localhost:3001/books')
      .then((response) => response.json())
      .then((data) => setBooks(data));
  }, []);

  return <List books={books} />;
}

export default App;
```

Listing 11.3 Loading the List Data from the Server and Displaying the List (src/App.tsx)

To load the data, you need a backend. For its implementation, you want to install the `json-server` package via the `npm install json-server` command. You also create a file named `data.json` in the root directory of your application. [Listing 11.4](#) shows the contents of this file:

```
{
  "books": [
    {
      "id": 1,
      "title": "JavaScript - The Comprehensive Guide",
      "author": "Philip Ackermann",
      "isbn": "978-3836286299",
      "rating": 5
    },
    {
      "id": 2,
      "title": "Clean Code",
      "author": "Robert Martin",
      "isbn": "978-0132350884",
      "rating": 4
    },
    {
      "id": 3,
      "title": "Design Patterns",
      "author": "Erich Gamma",
      "isbn": "978-0201633610",
      "rating": 5
    }
  ]
}
```

```
}  
]  
}
```

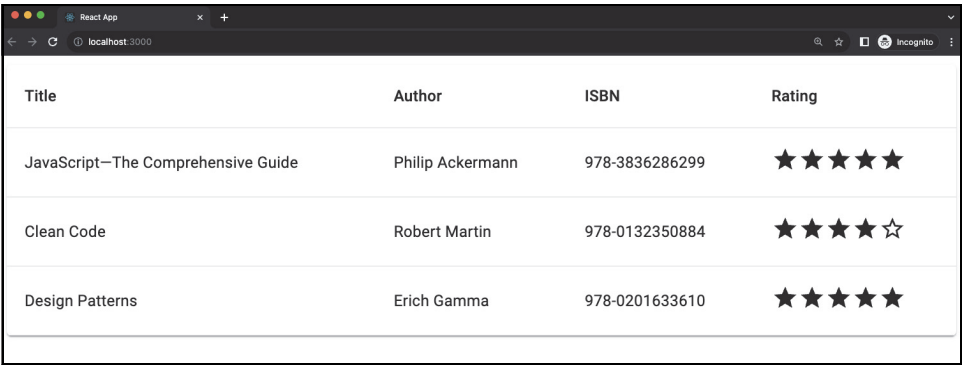
Listing 11.4 Data Source for the Backend (data.json)

You can either run the backend directly in the command line or extend the `scripts` section of your `package.json` file with an entry like the one shown in [Listing 11.5](#):

```
{  
  ...  
  "scripts": {  
    "start": "react-scripts start",  
    "build": "react-scripts build",  
    "test": "react-scripts test",  
    "eject": "react-scripts eject",  
    "backend": "json-server -p 3001 -w data.json"  
  },  
  ...  
}
```

Listing 11.5 Extract from the `package.json` File

This extension allows you to start your backend using the `npm run backend` command. In addition, you need to start the dev server of your frontend via `npm start` in a separate command line. Then you can view the current implementation of your list view in the browser. You should see a result like that shown in [Figure 11.1](#).



The screenshot shows a web browser window with the address bar set to `localhost:3000`. The browser displays a table with four columns: Title, Author, ISBN, and Rating. The table contains three rows of data. The first row is for 'JavaScript—The Comprehensive Guide' by Philip Ackermann with ISBN 978-3836286299 and a 5-star rating. The second row is for 'Clean Code' by Robert Martin with ISBN 978-0132350884 and a 4.5-star rating. The third row is for 'Design Patterns' by Erich Gamma with ISBN 978-0201633610 and a 5-star rating.

Title	Author	ISBN	Rating
JavaScript—The Comprehensive Guide	Philip Ackermann	978-3836286299	★★★★★
Clean Code	Robert Martin	978-0132350884	★★★★☆
Design Patterns	Erich Gamma	978-0201633610	★★★★★

Figure 11.1 Table Display of the Existing Data Records

11.2.1 Filtering the List in the Table

If the table in your application consists of only a handful of entries, the preceding implementation is sufficient. But if you have a large number of rows, you should

enable your users to filter the table to limit the number of rows displayed. The `Table` component doesn't provide any independent components for this purpose. Instead, it uses the standard React mechanisms. The basis of a filter is an input field that you implement as a *controlled component* (see [Chapter 10](#)). You use the component's state to filter the array of displayed records.

In the example, you can filter the table only by the **Title** column, so one input field is enough. If you want to filter by several columns, you either define several input fields, to each of which you assign a separate state structure and which you rate during display, or you filter all properties with only one input field and accordingly only one state.

The implementation of the filter function is up to you. You can start the search for a match at the beginning of the string or check for the mere presence of the searched string. You should also pay attention to uppercase and lowercase.

In the sample application, you enable filtering the table by title. Here you ignore the uppercase and lowercase. Also, the string you are looking for can be anywhere in the title. In the handler function of the change event, you use the `HTMLInputElement` type. You don't need to import this manually because TypeScript provides this type automatically. [Listing 11.6](#) shows the implementation of the table filter:

```
import React, { ChangeEvent, useState } from 'react';
import {
  Paper, Table, TableBody, TableCell, TableHead, TableRow, TextField,
} from '@mui/material';
import { StarBorder, Star } from '@mui/icons-material';
import { Book } from './Book';

type Props = {
  books: Book[];
};

const List: React.FC<Props> = ({ books }) => {
  const [filter, setFilter] = useState('');

  return (
    <Paper>
      <TextField
        label="Filter list"
        value={filter}
        onChange={({event: ChangeEvent<HTMLInputElement>}) =>
          setFilter(event.currentTarget.value)
        }
      />
```

```
<Table>
  <TableHead>
    <TableRow>
      <TableCell>Title</TableCell>
      <TableCell>Author</TableCell>
      <TableCell>ISBN</TableCell>
      <TableCell>Rating</TableCell>
    </TableRow>
  </TableHead>
  <TableBody>
    {books
      .filter((book) =>
        book.title.toLowerCase().includes(filter.toLowerCase())
      )
      .map((book) => (
        <TableRow key={book.id}>
          <TableCell>{book.title}</TableCell>
          <TableCell>{book.author}</TableCell>
          <TableCell>{book.isbn}</TableCell>
          <TableCell>
            {Array(5)
              .fill('')
              .map((rating, index) =>
                book.rating <= index ? (
                  <StarBorder key={index} />
                ) : (
                  <Star key={index} />
                )
              )}
          </TableCell>
        </TableRow>
      )]}
  </TableBody>
</Table>
</Paper>
);
};
```

```
export default List;
```

Listing 11.6 Implementation of the Table Filter (src/List.tsx)

Another standard requirement for lists and tables besides filtering is that users be able to sort them.

11.2.2 Sorting the Table

The display of the sorting direction is carried out by the `TableSortLabel` component of Material UI. However, the library doesn't provide any functionality for sorting the table itself. To implement sorting, you need to use the basic functionality of React and JavaScript. To be able to sort by all columns, you keep both the column to sort by and the direction to sort by in the state. Initially, you sort by title in ascending order and you pass this information as an initial value to the state hook. In the column headers, you integrate the `TableSortLabel` component per column and bind a click handler to it, which sets the information required for sorting in the state. [Listing 11.7](#) contains the source code required for sorting:

```
import React, { ChangeEvent, useState } from 'react';
import {
  Paper, Table, TableBody, TableCell, TableHead, TableRow, TextField,
  TableSortLabel,
} from '@mui/material';
import { StarBorder, Star } from '@mui/icons-material';
import { Book } from './Book';

type Props = {
  books: Book[];
};

const headers = {
  title: 'Title',
  author: 'Author',
  isbn: 'ISBN',
  rating: 'Rating',
};

const List: React.FC<Props> = ({ books }) => {
  const [filter, setFilter] = useState('');
  const [sort, setSort] = useState<{
    orderBy: keyof Book;
    order: 'asc' | 'desc';
  }>({
    orderBy: 'title',
    order: 'asc',
  });

  return (
    <Paper>
      <TextField
        label="Filter list"
```

```
    value={filter}
    onChange={({event}: ChangeEvent<HTMLInputElement>) =>
      setFilter(event.currentTarget.value)
    }
  />

<Table>
  <TableHead>
    <TableRow>
      {Object.entries(headers).map(([key, header]) => (
        <TableCell key={key}>
          <TableSortLabel
            active={sort.orderBy === key}
            direction={sort.order}
            onClick={() =>
              setSort({
                orderBy: key as keyof Book,
                order: sort.order === 'asc' ? 'desc' : 'asc',
              })
            }
          >
            {header}
          </TableSortLabel>
        </TableCell>
      ))}
    </TableRow>
  </TableHead>
  <TableBody>
    {books
      .filter((book) =>
        book.title.toLowerCase().includes(filter.toLowerCase())
      )
      .sort((a, b) => {
        const compareResult = a[sort.orderBy]
          .toString()
          .localeCompare(b[sort.orderBy].toString());
        return sort.order === 'asc' ? compareResult :
          -compareResult;
      })
      .map((book) => (
        <TableRow key={book.id}>
          <TableCell>{book.title}</TableCell>
          <TableCell>{book.author}</TableCell>
          <TableCell>{book.isbn}</TableCell>
```

```

      <TableCell>
        {Array(5)
          .fill('')
          .map((rating, index) =>
            book.rating <= index ? (
              <StarBorder key={index} />
            ) : (
              <Star key={index} />
            )
          )}
      </TableCell>
    </TableRow>
  )}
</TableBody>
</Table>
</Paper>
);
};

```

```
export default List;
```

Listing 11.7 Sorting the Table (src/List.tsx)

In this example, you can see one of the strengths of the Hook API of React. For sorting the table, you create a new state hook that contains all the information relevant for sorting. It's separated from filtering, so you have the option of swapping out sorting and filtering capabilities to their own components or to a custom hook.

As mentioned, the state hook receives the information about the initial sorting: this is done in ascending order based on the title. Unlike the previous example, you now create the table head in a loop. For this purpose, you define the `headers` object outside the component, which contains the field names as keys and the associated column headings as values. In the table head, you use the `Object.entries` method to get tuples of key and value. Then you apply the `map` method to this array and use a destructuring statement in the callback function to extract the key and label.

The `TableSortLabel` component takes care of displaying the sorting indicators as small arrow icons. For this purpose, you pass the `active` and `direction` props. The first, `active`, specifies whether sorting is currently performed according to the respective column; possible values are `true` or `false`. The `direction` prop determines the direction: `asc` for ascending and `desc` for descending. Finally, the third prop is the click handler, which triggers the actual sorting. Here you set the `sort` state to the current column and to an ascending or descending sorting.

You sort the table itself using the `array-sort` method. You combine this method with the existing `filter` method and pass the result to the `map` method. In the callback

function of the `sort` method, you use the `localeCompare` method of the JavaScript string type and make sure that the rating is also converted to a string. The `localeCompare` method returns the value 1, 0, or -1 to sort in the respective records. Depending on whether you want to sort in ascending or descending order, you must negate the result.

In [Figure 11.2](#), you can see the result of the adjustments to the list.

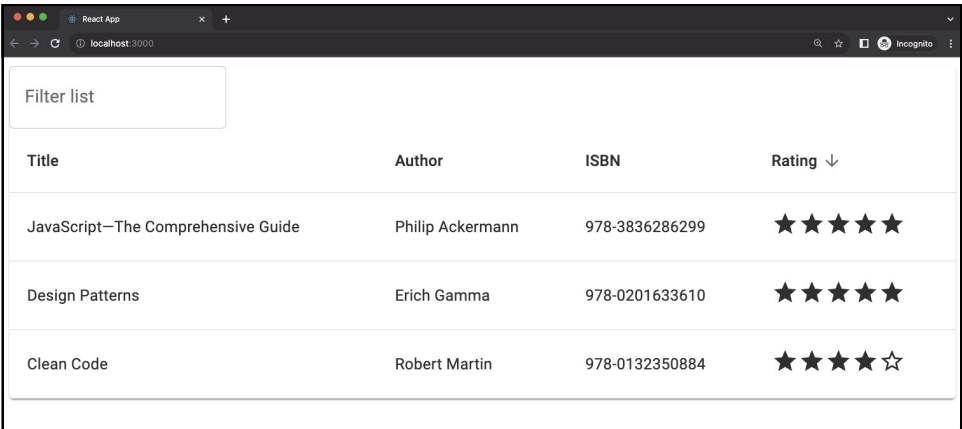


Figure 11.2 Sorting the List

In the next step, we'll look at how the application is displayed on different devices or on displays of different sizes.

11.3 Grids and Breakpoints

If you look at your application on a large display, the first thing you'll probably notice is that the table takes up the entire display width, making it look unnecessarily big. Spacing to the left and right of the table will solve this problem. But on a smaller display, such distances quickly become annoying because they limit the display area. This problem is usually solved by using a grid system.

Like many other UI libraries, the Material UI library provides its own grid system. The basis is a twelve-column grid with configurable column spacing. The grid system is based on CSS Flexbox. There's a total of five breakpoints defined for the Material UI grid, which you can see in [Table 11.1](#).

Breakpoint	Screen Width
xs	From 0 px
sm	From 600 px

Table 11.1 Breakpoints in Material UI

Breakpoint	Screen Width
md	From 960 px
lg	From 1,280 px
xl	From 1,920 px

Table 11.1 Breakpoints in Material UI (Cont.)

Like the underlying Flexbox, the Material UI grid uses containers and items. A *container* contains the *items*, which can have a width from 1 to 12 columns. Both containers and items are represented by the `Grid` component. To render a container, you specify the `container` prop; for an item, you need to specify the `item` prop. A container component also accepts the `spacing` prop, to which you can pass a numeric value. This value specifies the distance between columns as a multiple of 8 px. For the individual items, you can use the breakpoint names as props and specify how many columns each breakpoint should take up.

For the sample application, you choose a three-column layout in the `md` and larger breakpoints. The left and right columns each take up one grid column, leaving 10 columns for the content. This in turn takes up all available 12 columns in the `xs` and `sm` breakpoints. Thus, there is no room left for the two remaining columns with this display size. In this case, the grid opens a new row, which means that an empty element is rendered above and below the table. You can fix this problem by applying a custom style and media query to the two columns. Here you have a very fine control option to determine when and which styling properties should be applied.

```
import React, { ChangeEvent, useState } from 'react';
import { Grid, Hidden, Paper, Table, TableBody, TableCell, TableHead, TableRow,
  TextField, TableSortLabel } from '@mui/material';
import { StarBorder, Star } from '@mui/icons-material';
import { Book } from './Book';

type Props = {
  books: Book[];
};

const headers = {
  title: 'Title',
  author: 'Author',
  isbn: 'ISBN',
  rating: 'Rating',
};
```

```
const List: React.FC<Props> = ({ books }) => {
  const [filter, setFilter] = useState('');
  const [sort, setSort] = useState<{
    orderBy: keyof Book;
    order: 'asc' | 'desc';
  }>({
    orderBy: 'title',
    order: 'asc',
  }));

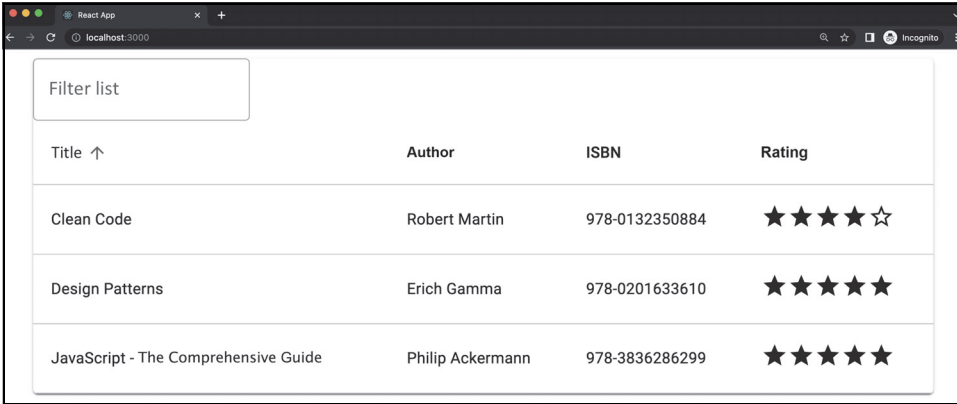
  return (
    <Grid container>
      <Grid item md={1} sx={{ display: { sm: 'none', md: 'block' } }} />
      <Grid item xs={12} md={10}>
        <Paper>
          <TextField
            label="Filter list"
            value={filter}
            onChange={(event: ChangeEvent<HTMLInputElement>) =>
              setFilter(event.currentTarget.value)
            }
          />

          <Table>...</Table>
        </Paper>
      </Grid>
      <Grid item md={1} sx={{ display: { sm: 'none', md: 'block' } }} />
    </Grid>
  );
};

export default List;
```

Listing 11.8 Using the Material UI Grid (src/admin/List.tsx)

You can proceed in the same way with the grid columns as with many other Material UI components (e.g., entire table columns). If the table becomes too wide for you on smaller screens and you want to do without a horizontal scrollbar, you can hide individual columns. As an alternative to using tables, you can also use a simple list display for the presentation on mobile devices and render the table or the list respectively. In [Figure 11.3](#), you can see the result of using the `Grid` component on a wide screen.



Title ↑	Author	ISBN	Rating
Clean Code	Robert Martin	978-0132350884	★★★★☆
Design Patterns	Erich Gamma	978-0201633610	★★★★★
JavaScript - The Comprehensive Guide	Philip Ackermann	978-3836286299	★★★★★

Figure 11.3 “Grid” Component on a Wide Screen

The display of information in tables is only one aspect of Material UI. The next step is to enable your users to interact with your application via components.

11.4 Icons

Users have numerous options for interacting with a web application. The most obvious one is the use of form elements such as input fields and buttons. But you can also use links to let users interact with your application. Furthermore, you can bind various event listeners to any element of your application, allowing you to interact with these elements.

At this point, it’s important that such an interactive element is also recognizable as such for the users. The shape and behavior of the elements are of crucial importance. Users are familiar with form elements that can be used intuitively. Links are usually highlighted by the browser—for example, with a color that differs from regular body text or with underlining. In addition, a link responds to the touch of the mouse cursor by changing the style slightly and changing the shape of the mouse cursor. These features are missing when you add an event listener to a standard element. So here you have to make sure yourself that the users recognize the interaction option.

In web applications, icons are often used for interactive elements as they visually describe the interaction option. So instead of inserting a link with the word *edit* or *delete*, you can use a pencil icon or a trash can icon respectively. These icons are used so frequently that users understand the purpose without further explanation. To further improve usability at this point, you should use alternative texts that are displayed on mouseover and briefly describe the action in words. You can further improve the visualization of the interaction option for icons by using a button or a link element as a container.

At the start of this chapter, you installed the `@mui/icons-material` package. It contains a variety of icons that you can use in your application. The icons are available as SVG images and can therefore be styled and scaled using CSS. In the following code example, you'll add icons to delete data records in the table. For this purpose, you need the two corresponding icons from the `@mui/icons-material` package. You also use the `IconButton` component, which is responsible for the visualization of the interaction option. You can see the modifications to the `List` component in [Listing 11.9](#):

```
import React, { ChangeEvent, useState } from 'react';
import { Grid, Paper, Table, TableBody, TableCell, TableHead, TableRow,
  TextField, TableSortLabel, IconButton } from '@mui/material';
import { StarBorder, Star, Delete } from '@mui/icons-material';
import { Book } from './Book';

type Props = {
  books: Book[];
  onDelete: (book: Book) => void;
};

const headers = {...};

const List: React.FC<Props> = ({ books, onDelete }) => {
  const [filter, setFilter] = useState('');
  const [sort, setSort] = useState<{...}>({...});

  return (
    <Grid container>
      <Grid item md={1} sx={{ display: { sm: 'none', md: 'block' } }} />
      <Grid item xs={12} md={10}>
        <Paper>
          <TextField
            label="Filter list"
            value={filter}
            onChange={(event: ChangeEvent<HTMLInputElement>) =>
              setFilter(event.currentTarget.value)
            }
          />

          <Table>
            <TableHead>
              <TableRow>
                {Object.entries(headers).map(([key, header]) => (
                  <TableCell key={key}>...</TableCell>
                ))}
```

```

        <TableCell />
      </TableRow>
    </TableHead>
    <TableBody>
      {books
        .filter((book) => ...)
        .sort((a, b) => {...})
        .map((book) => (
          <TableRow key={book.id}>
            <TableCell>{book.title}</TableCell>
            ...
            <TableCell>
              <IconButton
                color="primary"
                aria-label="delete book"
                onClick={() => onDelete(book)}
              >
                <Delete />
              </IconButton>
            </TableCell>
          </TableRow>
        )}}
    </TableBody>
  </Table>
</Paper>
</Grid>
<Grid item md={1} sx={{ display: { sm: 'none', md: 'block' } }} />
</Grid>
);
}

```

```
export default List;
```

Listing 11.9 Integrating the “IconButton” for Deletions (src/List.tsx)

In the next step, you’ll ensure that the button you’ve just integrated can actually be used.

11.5 Deleting Data Records

The data shown in the table is currently located in the `App` component. This means that you should take care of the server communication and removal of data records here. The `onDelete` function, which is responsible for deleting, is passed to the list via props.

In the list, you then bind the click handler of the **Delete** button to the function; as a result, the functionality is implemented. However, it's common practice for delete operations to ask the user if they're really sure before they finally remove the data record. For this request, you can create a dialog box where the user either confirms or cancels the delete operation. The implementation is done via the `Dialog` component of Material UI.

11.5.1 Preparing a Delete Operation

Before you integrate the `Dialog` component, you first need to implement the actual delete function in the `App` component. For this purpose, you create a `handleDelete` function as shown in [Listing 11.10](#):

```
import React, { useEffect, useState } from 'react';
import './App.css';
import { Book } from './Book';
import List from './List';

const App: React.FC = () => {
  const [books, setBooks] = useState<Book[]>([]);

  useEffect(() => {
    fetch('http://localhost:3001/books')
      .then((response) => response.json())
      .then((data) => setBooks(data));
  }, []);

  async function handleDelete(book: Book) {
    fetch(
      `http://localhost:3001/books/${book.id}`,
      { method: 'DELETE' }
    ).then(
      (response) => {
        if (response.ok) {
          setBooks((prevBooks) =>
            prevBooks.filter((prevBook) => prevBook.id !== book.id)
          );
        }
      }
    );
  }

  return <List books={books} onDelete={handleDelete} />;
}
```

```
};
```

```
export default App;
```

Listing 11.10 Function to Delete Data Records (src/App.tsx)

The asynchronous `handleDelete` function is responsible for deleting the data record and accepts the object representation of the corresponding record for this purpose. The function uses the `id` property to generate the URL that identifies the resource on the server. Using the Fetch API of the browser, you issue a `DELETE` request. After that, you remove the record from the state, which will cause the display to be updated. You pass the `handleDelete` function to the list in the `onDelete` prop.

In a production application, you must provide routines for error handling at this point. For example, if the server request fails, you should notify the user of this. This is often done by displaying a corresponding error message.

11.5.2 Implementing a Confirmation Dialog

The `Dialog` component provides the basic styling as well as the functionality to open and close dialogs. There are also other components you can use to define other elements of such a dialog. Examples include the title, the actual content, and the action buttons. The dialog is essentially controlled via the `open` prop of the `Dialog` component. If its value is `true`, the dialog is visible. If it's `false`, the dialog is closed.

Dialogs are a typical example of reusable components. Basically, a dialog has nothing to do with the component in which it's used. The dialog that you use to confirm the deletion of a data record in a list shouldn't need to know anything about the `List` component except for the display text. The integrating `List` component passes the information about whether the dialog should be displayed, which text you want to display, and what should happen after the users' interaction. This way you keep the dialog general so you can reuse it. The actual implementation of the dialog is shown in [Listing 11.11](#). Save this source code in the `ConfirmDialog.tsx` file in the `src` directory.

```
import React from 'react';
import { Dialog, DialogTitle, DialogContent, DialogContentText, DialogActions, Button } from '@mui/material';

type Props = {
  open: boolean;
  title: string;
  text: string;
  onConfirm: (confirmation: boolean) => void;
}
```



```
const ConfirmDialog: React.FC<Props> = ({ onConfirm, title, text, open }) => {
  return (
    <Dialog
      open={open}
      onClose={() => onConfirm(false)}
      aria-labelledby="confirm-dialog-title"
      aria-describedby="confirm-dialog-description"
    >
      <DialogTitle id="confirm-dialog-title">{title}</DialogTitle>
      <DialogContent>
        <DialogContentText id="confirm-dialog-description">
          {text}
        </DialogContentText>
      </DialogContent>
      <DialogActions>
        <Button onClick={() => onConfirm(false)} color="secondary">
          Cancel
        </Button>
        <Button onClick={() => onConfirm(true)} color="primary" autoFocus>
          OK
        </Button>
      </DialogActions>
    </Dialog>
  );
};

export default ConfirmDialog;
```

Listing 11.11 Confirmation Dialog (src/ConfirmDialog.tsx)

In the `ConfirmDialog` component, you display the `title` prop as the heading of the dialog and the `text` as the content. The `open` prop determines whether the dialog is open or closed. The `onClose` prop of the `Dialog` component makes sure that the `onConfirm` function is executed with the value `false` as soon as a user presses the `[Esc]` key or clicks somewhere outside the dialog. The two buttons of the dialog also execute the `onConfirm` function: the **Cancel** button with the value `false`, the **OK** button with the value `true`.

The two `aria` attributes improve usability for people with disabilities who rely on a screen reader, for example. The two `DialogTitle` and `DialogContent` components are largely self-explanatory. In the `DialogActions` component, you insert two `Button` components. The first button should cancel the respective action and the second button should confirm it. At this stage of the implementation, the only thing missing is the inclusion in the list—and then you'll be able to delete data records.

11.5.3 Deleting Data Records

In the `List` component, you first create a local state that stores which record you have marked for deletion and whether the confirmation dialog is open. Then you create a click handler for the **Delete** button, which marks the corresponding record for deletion and opens the dialog. Finally, you integrate the `ConfirmDialog` component and link it to the local state:

```
import React, { ChangeEvent, useState } from 'react';
import { Grid, Paper, Table, TableBody, TableCell, TableHead, TableRow, TextField, TableSortLabel, IconButton } from '@mui/material';
import { StarBorder, Star, Delete } from '@mui/icons-material';
import { Book } from './Book';
import ConfirmDialog from './ConfirmDialog';

type Props = {...};
const headers = {...};

const List: React.FC<Props> = ({ books, onDelete }) => {
  const [deleteDialog, setDeleteDialog] = useState<{
    open: boolean;
    book: Book | null;
 }>({ open: false, book: null });
  const [filter, setFilter] = useState('');
  const [sort, setSort] = useState<{...}>({...});

  return (
    <Grid container>
      <Grid item md={1} sx={{ display: { sm: 'none', md: 'block' } }} />
      <Grid item xs={12} md={10}>
        <Paper>
          <TextField ... />

          <Table>
            <TableHead>...</TableHead>
            <TableBody>
              {books
                .filter((book) => ...)
                .sort((a, b) => {...})
                .map((book) => (
                  <TableRow key={book.id}>
                    <TableCell>...</TableCell>
                    <TableCell>
                      <IconButton
                        color="primary"

```

```

        aria-label="delete book"
        onClick={() => {
          setDeleteDialog({ open: true, book: book });
        }}
      >
        <Delete />
      </IconButton>
    </TableCell>
  </TableRow>
)}}
```

```

  </TableBody>
</Table>
</Paper>
</Grid>
<Grid item md={1} sx={{ display: { sm: 'none', md: 'block' } }} />
<ConfirmDialog
  title="Really delete?"
  text="Are you sure you want to delete the selected item?"
  open={deleteDialog.open}
  onConfirm={(confirmation) => {
    if (confirmation && deleteDialog.book) {
      onDelete(deleteDialog.book);
    }
    setDeleteDialog({
      open: false,
      book: null,
    });
  }}
/>
</Grid>
);
};

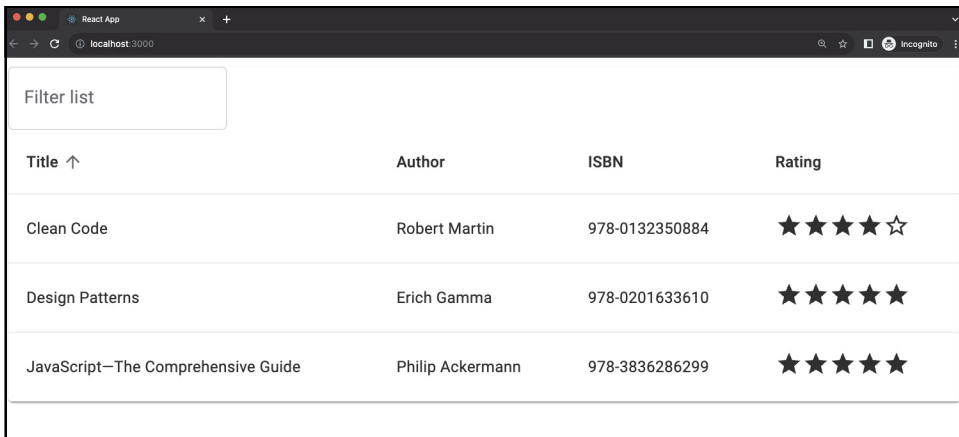
export default List;
```

Listing 11.12 Integration of the Confirmation Dialog (src/admin/List.tsx)

At the beginning of the `List` component, which you can find in [Listing 11.12](#), you generate a separate state for the confirmation dialog. It consists of a Boolean property named `open` for the state of the dialog and a `book` property of type `Book | null`, which contains a reference to the data record to be deleted. In the click handler of the **Delete** button, you set the `open` property of the `deleteDialog` state to `true` and reference the affected record. This ensures that the dialog is displayed.

At the end of the file—still inside the root element—you integrate the `ConfirmDialog` component. Then you pass the title and the text to be displayed as well as the reference to the state of the dialog—that is, whether or not it should be displayed. Once you’ve made these adjustments, you can switch to your browser and start deleting the existing data records. The result should look like the one shown in [Figure 11.4](#). If you click the **Cancel** button, the record will be kept; if you press the **OK** button instead, the record will be deleted both on the server and in the view in the browser.

Once you’ve tested your newly implemented delete feature, you must be able to integrate new data records.



Title ↑	Author	ISBN	Rating
Clean Code	Robert Martin	978-0132350884	★★★★☆
Design Patterns	Erich Gamma	978-0201633610	★★★★★
JavaScript—The Comprehensive Guide	Philip Ackermann	978-3836286299	★★★★★

Figure 11.4 Confirmation Dialog

11.6 Creating New Data Records

For the creation, as with the deletion, you first need a routine in the `App` component that handles the server communication. Then you create a `Form` component based on the React Hook Form library so that the form can be opened in a dialog and will conform to the Material layout. In the final step, you integrate both the routine from the `App` component and the form in the `List` component.

11.6.1 Preparing the Creation of Data Records

Currently, the `App` component is responsible for data management. Thus, similar to the delete routine, you add a new `handleSave` function that takes care of saving the data by using the Fetch API. You pass this function to the `List` component via the `onSave` prop. The source code of the customized `App` component can be found in [Listing 11.13](#):

```
import React, { useEffect, useState } from 'react';
import './App.css';
import { Book } from './Book';
```

```
import List from './List';

const App: React.FC = () => {
  const [books, setBooks] = useState<Book[]>([]);

  useEffect(() => {...}, []);

  async function handleDelete(book: Book) {...}

  async function handleSave(book: InputBook) {
    const request = await fetch('http://localhost:3001/books', {
      method: 'POST',
      body: JSON.stringify(book),
      headers: { 'content-type': 'application/json' },
    });
    const data = await request.json();
    setBooks((prevBooks) => [...prevBooks, data]);
  }

  return <List books={books} onDelete={handleDelete} onSave={handleSave} />;
};

export default App;
```

Listing 11.13 The “handleSave” Function in the “App” Component (src/App.tsx)

The `handleSave` function accepts a data record and sends it to the server along with a POST request. The server responds with the newly added data record to which a new ID has been added. You integrate this data record into the state so that React updates the component and displays the record.

11.6.2 Implementation of the “Form” Component

The `Form` component is a combination of a dialog and a form. For this purpose, you integrate the `Dialog`, `DialogTitle`, `DialogContent`, and `DialogActions` components. In addition, you insert the `TextField` components from Material UI. You can see the source code of the `Form.tsx` file that contains the `Form` component in [Listing 11.14](#). We’ll go through it step by step.

```
import React, { useEffect } from 'react';
import validationSchema from './validationSchema';
import { yupResolver } from '@hookform/resolvers/yup';
import {
  Dialog,
  DialogTitle,
```

```

    DialogContent,
    DialogActions,
    Button,
    TextField,
  } from '@mui/material';
import { InputBook } from './Book';
import { useForm } from 'react-hook-form';

interface Props {
  open: boolean;
  book?: InputBook;
  onSave: (book: InputBook) => void;
  onClose: () => void;
}

const Form: React.FC<Props> = ({
  open,
  book = { title: '', author: '', isbn: '' },
  onSave,
  onClose,
}) => {
  const {
    register,
    handleSubmit,
    reset,
    formState: { errors },
  } = useForm<InputBook>({
    defaultValues: book,
    resolver: yupResolver(validationSchema),
  });
  useEffect(() => {
    if (book.id) {
      reset(book);
    }
  }, [book, reset]);
  return (
    <Dialog
      open={open}
      onClose={onClose}
      aria-labelledby="form-dialog-title"
      aria-describedby="form-dialog-description"
    >
      <DialogTitle id="form-dialog-title">
        {book.id ? 'Edit book' : 'Create new book'}
      </DialogTitle>
    </Dialog>
  );
};

```

```
</DialogTitle>
<form onSubmit={handleSubmit(onSave)}>
  <DialogContent id="form-dialog-description">
    <div>
      <TextField
        {...register('title')}
        error={!!errors.title}
        label="Title"
      />
      {errors.title && (
        <div style={{ color: 'red' }}>{errors.title.message}</div>
      )}
    </div>
    <div>
      <TextField
        {...register('author')}
        error={!!errors.author}
        label="Author"
      />
      {errors.author && (
        <div style={{ color: 'red' }}>{errors.author.message}</div>
      )}
    </div>
    <div>
      <TextField
        {...register('isbn')}
        error={!!errors.isbn}
        label="ISBN"
      />
      {errors.isbn && (
        <div style={{ color: 'red' }}>{errors.isbn.message}</div>
      )}
    </div>
  </DialogContent>
  <DialogActions>
    <Button color="secondary" onClick={onClose}>
      Cancel
    </Button>
    <Button color="primary" type="submit">
      Saving
    </Button>
  </DialogActions>
</form>
</Dialog>
```

```
);
};

export default Form;
```

Listing 11.14 “Form” Component with Material UI Components (src/Form.tsx)

The `Form` component accepts the `open` and `onClose` props for the dialog control. You also define an optional `book` prop, which allows you to pass a data record for editing, and an `onSave` function that the component executes when the form is submitted.

The opening tag of the `Dialog` component is followed directly by the `form` element, whose `onSubmit` you connect to the `handleSubmit` function of `React Hook Form` and the `onSave` function from the props. Inside the form, the dialog components with the title and the content follow. This kind of structuring is necessary so that you can place the buttons in the `DialogActions` component.

You can easily combine the `register` function from `React Hook Form` and the form components from `Material UI`. Both libraries follow the standard, so you don't have to expect any problems. They also give you the flexibility to control virtually every aspect of your form. You can render not only input elements, but also custom components. Using the `formState.errors` property, you can also integrate the appropriate error-handling process into your form right away.

The final part of the `Form` component is the `DialogActions` component with the buttons for canceling—that is, closing—the dialog and for saving. For the form to work properly, you also need a validation schema, which you store in the `validationSchema.ts` file in the `src` directory. You can see the structure of this file in [Listing 11.15](#):

```
import * as Yup from 'yup';

const validationSchema = Yup.object().shape({
  title: Yup.string().required('Title is a required field.'),
  author: Yup.string().required('Author is a required field.'),
  isbn: Yup.string().required('ISBN is a required field'),
});

export default validationSchema;
```

Listing 11.15 Validation Schema for the Form (src/validationSchema.ts)

11.6.3 Integration of the Form Dialog

In the `List` component, you need a local state that contains the state of the form dialog. You'll also integrate a button that will help you open the dialog. The corresponding source code is shown in [Listing 11.16](#):


```
import React, { ChangeEvent, useState } from 'react';
import { Grid, Paper, Table, TableBody, TableCell, TableHead, TableRow,
  TextField, TableSortLabel, IconButton, Fab } from '@mui/material';
import { StarBorder, Star, Delete, Add } from '@mui/icons-material';
import { Book, InputBook } from './Book';
import ConfirmDialog from './ConfirmDialog';
import Form from './Form';

type Props = {...};

const headers = {...};

const List: React.FC<Props> = ({ books, onDelete, onSave }) => {
  const [deleteDialog, setDeleteDialog] = useState<{...}>({...});
  const [filter, setFilter] = useState('');
  const [sort, setSort] = useState<{...}>({...});
  const [formDialog, setFormDialog] = useState<boolean>(false);

  return (
    <Grid container>
      <Grid item md={1} sx={{ display: { sm: 'none', md: 'block' } }} />
      <Grid item xs={12} md={10}>...</Grid>
      <Grid item md={1} sx={{ display: { sm: 'none', md: 'block' } }} />
      <ConfirmDialog... />
      <Form
        onSave={(book: InputBook) => {
          setFormDialog(false);
          onSave(book);
        }}
        open={formDialog}
        onClose={() => setFormDialog(false)}
      />
      <Fab
        color="primary"
        aria-label="Add"
        onClick={() => {
          setFormDialog(true);
        }}
      />
      <Add />
    </Grid>
  );
};
```

```
};
```

```
export default List;
```

Listing 11.16 Integration of the Form Dialog (src/List.tsx)

Using the state hook, you define a separate state for the form dialog—that is, whether it is open or closed. You pass this information to the `Form` component via the `open` prop. In addition, the form gets a `save` and a `close` handler. You implement the button to open the dialog in the form of the `Fab` component, where *fab* means *floating action button*. It's used for global actions, such as adding records in a view of an application. Such a fab is usually placed centrally at the bottom of the screen. Clicking the button opens the form dialog by setting the `formDialog` state to `true`. The final step consists of making sure that the fab is actually positioned as advertised.

Positioning the Fab

The positioning of the fab within the `List` component is the responsibility of the corresponding stylesheet or the file containing the styled components. The name of the file is `List.styles.ts`, and it's located in the `src` directory. The contents of this file are shown in [Listing 11.17](#):

```
import { Fab as MatFab } from '@mui/material';
import styled from '@emotion/styled';

export const Fab = styled(MatFab)`
  && {
    position: fixed;
    bottom: 0;
    left: 50%;
    margin-bottom: 20px;
    transform: translateX(-50%);
  }
`;
```

Listing 11.17 Positioning the Fab (src/List.styles.ts)

In this code example, you see that you can use a combination of Emotion and the components from Material UI Styles to customize the default styles of the components according to your requirements. To make sure the modified version of the `Fab` component is activated, you replace the import of the `Fab` component in the `List` component with the styled-component version of the fab. Then you can return to your browser and test adding new data records. The display should look somewhat like [Figure 11.5](#).

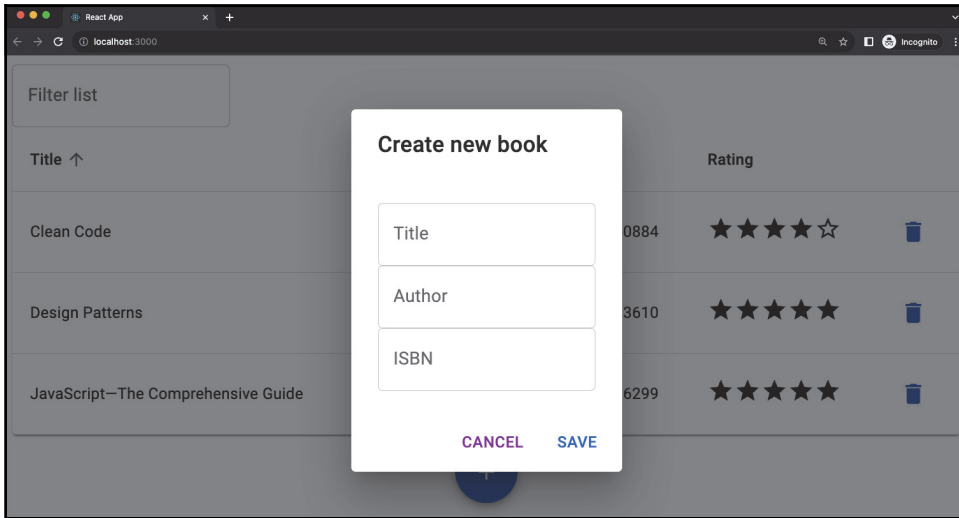


Figure 11.5 Adding New Data Records (src/List.styles.ts)

So that you can fully manage the data records in your application, now that you can delete and create records, the only thing missing is an option to edit existing records.

11.7 Editing Data Records

The `Form` component is already prepared for editing records. For this reason, no further adjustments are required, and you can focus on the `List` and `App` components. In addition to the state of the dialog, you need to store the data record to be changed in the `formDialog` state of the `List` component. Furthermore, next to the **Delete** button, you need to add an **Edit** button with a corresponding click handler that makes sure that the `formDialog` state is set accordingly. In the final step, you still need to pass the data record to the `Form` component. [Listing 11.18](#) contains the adjusted source code of the `List` component:

```
import React, { ChangeEvent, useState } from 'react';
import { Grid, Paper, Table, TableBody, TableCell, TableHead, TableRow,
  TextField, TableSortLabel, IconButton } from '@mui/material';
import { StarBorder, Star, Delete, Add, Edit } from '@mui/icons-material';
import { Book, InputBook } from './Book';
import ConfirmDialog from './ConfirmDialog';
import Form from './Form';
import { Fab } from './List.styles';
```

```
type Props = {};
```

```
const headers = {};
```

```

const List: React.FC<Props> = ({ books, onDelete, onSave }) => {
  const [deleteDialog, setDeleteDialog] = useState<{...}>({...});
  const [filter, setFilter] = useState('');
  const [sort, setSort] = useState<{...}>({...});
  const [formDialog, setFormDialog] = useState<{
    open: boolean; book?: Book
  }>({ open: false });

  return (
    <Grid container>
      <Grid item md={1} sx={{ display: { sm: 'none', md: 'block' } }} />
      <Grid item xs={12} md={10}>
        <Paper>
          <TextField label="Liste filtern" ... />

          <Table>
            <TableHead>
              <TableRow>
                {Object.entries(headers).map(([key, header]) => ( ... ))}
                <TableCell />
                <TableCell />
              </TableRow>
            </TableHead>
            <TableBody>
              {books
                .filter((book) => ... )
                .sort((a, b) => { ... })
                .map((book) => (
                  <TableRow key={book.id}>
                    <TableCell>{book.title}</TableCell>
                    <TableCell>{book.author}</TableCell>
                    <TableCell>{book.isbn}</TableCell>
                    <TableCell>...</TableCell>
                    <TableCell>
                      <IconButton ... >
                        <Delete />
                      </IconButton>
                    </TableCell>
                    <TableCell>
                      <IconButton
                        color="primary"
                        aria-label="edit book"
                        onClick={() => {
                          setFormDialog({ open: true, book: book });

```

```
        }}
      >
        <Edit />
      </IconButton>
    </TableCell>
  </TableRow>
</Table>
</TableBody>
</Table>
</Paper>
</Grid>
<Grid item md={1} sx={{ display: { sm: 'none', md: 'block' } }} />
<ConfirmDialog ... />
<Form
  onSave={book: InputBook} => {
    setFormDialog({ open: false, book: undefined });
    onSave(book);
  }}
  book={formDialog.book}
  open={formDialog.open}
  onClose={() => setFormDialog({ open: false, book: undefined })}
/>
<Fab
  color="primary"
  aria-label="Add"
  onClick={() => {
    setFormDialog({ open: true, book: undefined });
  }}
>
  <Add />
</Fab>
</Grid>
);
};

export default List;
```

Listing 11.18 Integration of the Editing Routine in the List (src/List.tsx)

The `formDialog` state gets the optional `book` property, which contains the record to be changed. Within the click handler of the **Edit** button, you set both the `open` state of the dialog so that it is displayed and the `book` property so that the form is prepopulated with the appropriate values. In the final step, you pass the data record from the `formDialog` state to the `Form` component.

In the App component, which has so far only taken care of storing new data records, you have to distinguish whether a record is a new or an existing record. An update is done using the PUT value of the method property in the Fetch API. Such a request requires appending the ID of the data record to be modified to the URL path. Also, you need to replace the existing record in the books array instead of simply appending the data record returned by the server. [Listing 11.19](#) shows the source code of the adjusted App component:

```
import React, { useEffect, useState } from 'react';
import './App.css';
import { Book, InputBook } from './Book';
import List from './List';

const App: React.FC = () => {
  const [books, setBooks] = useState<Book[]>([]);

  useEffect(() => {...}, []);

  async function handleDelete(book: Book) {...}

  async function handleSave(book: InputBook) {
    let url = 'http://localhost:3001/books';
    let method = 'POST';

    if (book.id) {
      method = 'PUT';
      url += `/${book.id}`;
    }

    const request = await fetch(url, {
      method,
      body: JSON.stringify(book),
      headers: { 'content-type': 'application/json' },
    });
    const data = await request.json();
    if (book.id) {
      setBooks((prevBooks) =>
        books.map((prevBook) => {
          if (prevBook.id === book.id) {
            return data;
          }
          return prevBook;
        })
      );
    }
  }
};
```

```

    } else {
      setBooks((prevBooks) => [...prevBooks, data]);
    }
  }

  return <List books={books} onDelete={handleDelete} onSave={handleSave} />;
};

export default App;

```

Listing 11.19 Saving the Modified Data Record (src/admin/Admin.tsx)

If you edit an existing data record, it will have an ID. Based on this criterion, you can distinguish how to handle the dataset. For existing records, you use the PUT value for the method property in the fetch call; for new records, you use POST as the value. To indicate the modification to a data record, you use the map method of the books array within the state setter and return the modified record here instead of the original version of the record.

What's left is the handling of new data records, so you can move this routine to the else branch without any adjustment.

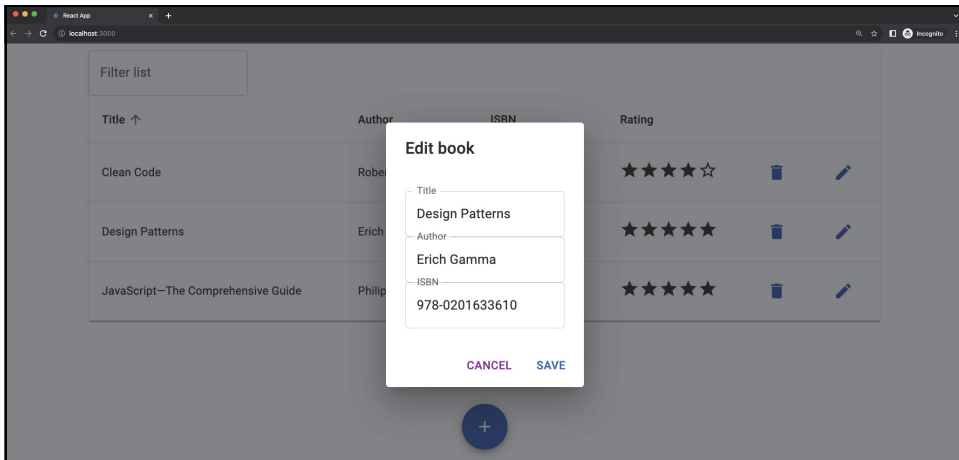


Figure 11.6 Editing an Existing Data Record

Clicking on the pencil icon in the browser next to a data record will open that record in the form (see [Figure 11.6](#)) and allow you to adjust and save the values.

11.8 Summary

In this chapter, you learned about the integration of the Material UI component library. Specifically, you learned the following:

- Using the `Table` component of Material UI, you can achieve many different representations of table values.
- You learned about two important features: sorting and filtering tables. When implementing these features, you can mainly rely on JavaScript and are supported by the component library for visualization.
- The grid system that comes with Material UI allows you to optimize the layout of your application to different display sizes.
- Material UI has a comprehensive icon library that you can use in your application to mark certain actions intuitively and with little explanation.
- Components like the `Dialog` component make it easy for you to use standard elements in web development. These components can be adjusted to the look and feel of your application using stylesheets.
- Material UI offers a wide range of form components. These range from simple input fields and buttons to checkboxes and dropdowns.

In the next chapter, you'll learn how to integrate the individual components of an application using the React router so that you can switch between the individual views of the application.

Chapter 12

Navigating Within an Application: The Router

A *single-page application* (SPA) rarely consists of only one view. As its name suggests, an SPA doesn't offer a way to keep the individual views on independent pages and switch between them. There are several reasons why this is the case:

- A page change means that the browser must reload the entire application, including the framework and all dependencies. During the build process, these resources are combined into one file, but still an additional request to the server always means that a waiting time for your users exists.
- The second reason against page switching in the SPA context is that loading a new page loses the state of the application in the memory. In this case, the browser discards all objects and you have to rebuild them. This costs time, and there is a risk of losing nonpersisted information,

This leads to the requirement that it should be possible to switch between different views of an application without causing the browser to load the entire page. In most modern solutions, this can be implemented in such a way that it appears to users that the desired view is still selected via a standalone URL. You'll learn how this works and how you can integrate it into your application in this chapter.

React itself doesn't support *routing*—as the navigation is referred to. This means that you will have to resort to a solution in the form of another package. The default package for routing in React is named *React Router*. The project is developed independently of React and can be included via a package manager of your choice.

12.1 Installation and Integration

React Router is being developed as an open-source project on GitHub. The official website of the project can be found at <https://reactrouter.com/> and the GitHub repository at <https://github.com/remix-run/react-router>. The router is developed as a monorepo and consists of several packages. Thus, there is a variant for the web and another for native apps implemented using React Native. In the following sections, we'll deal with the web variant, which can be installed via the `npm install react-router-dom@6` command. For use with TypeScript, you don't need to install any additional type definitions: React

Router is completely implemented in TypeScript, and therefore no external definitions are required.

The React Router package you just installed contains numerous components that you can use to implement the navigation feature in your application. The two most important components are `BrowserRouter` and `HashRouter`, which form the basis of React Router. The reason for two different components is so that you can choose between two different technologies for navigation.

The `BrowserRouter` component is based on the HTML5 History API. The most important elements here are the `pushState` method, which allows you to set a new history state, and the `popstate` event, which is triggered when the active history state is changed. The browser displays the states of the History API via the URL path in the address bar. Besides setting the path, you can pass additional information. Adjusting the history state looks to users as if they have changed pages. The URL changes, and the router responds to the state change by rendering a new component or an entire component tree. You don't need to bother with accessing the History API yourself as the React Router components take care of that for you.

The modified URL path allows your users to use this path as an entry point to your application as well. For this reason, you should also configure the web server to forward all requests received at the frontend to *index.html*. For example, for an Nginx web server, such a redirect looks as follows:

```
location / {  
    try_files $uri /index.html;  
}
```

Listing 12.1 Redirect for React Router on Nginx

The second basic component of React Router, the `HashRouter` component, works similarly to the `BrowserRouter` component. The difference lies in the basic technology used. The `HashRouter` component doesn't rely on the already presented History API, but on *hash navigation*. This term refers to the hash part of the URL. Originally it was used for addressing skip marks in HTML documents. In the URL <https://reactjs.org/docs/components-and-props.html#function-and-class-components>, for example, *#function-and-class-components* is the hash part. If you're on a page, changing the hash part of the URL doesn't cause the page to load. Instead, the browser tries to navigate to the specified skip mark. In this process, the `hashchange` event is triggered, and `HashRouter` responds.

As a rule, you should rely on `BrowserRouter`, as the History API is the more modern and elegant solution. No matter which of the two variants you choose, both behave in the same way when they're integrated into the application and differ only in the display of the navigation path in the address bar.

12.2 Navigating in the Application

Quick Start

React Router follows a declarative approach. This means that you describe the routing system of your application in terms of components. The `BrowserRouter` component is the basis. Inside this component, you can integrate one or several `Routes` components. Finally, you define the actual routes using `Route` components. These accept a `path` prop, which you use to specify a URL path. If this path matches the current location of the browser, React Router renders the component you specified in the `element` prop.

You usually integrate the `BrowserRouter` component at a central point in the application. Within the router, you then define the individual routes using the `Route` component. For each route, you use props to specify the path to which the route should respond and the component that the router should render. As an example, you'll now implement routes to specific components in an application. For this purpose, you first include the `BrowserRouter` component in the `App` component and define the routes for the `List` and `Form` components. [Listing 12.2](#) contains the corresponding source code:

```
import React from 'react';
import { BrowserRouter, Routes, Route, Navigate } from 'react-router-dom';
import List from './List';
import Form from './Form';

const App: React.FC = () => {
  return (
    <BrowserRouter>
      <Routes>
        <Route path="/list" element={<List />} />
        <Route path="/form" element={<Form />} />
        <Route path="/" element={<Navigate to="/list" />} />
      </Routes>
    </BrowserRouter>
  );
};

export default App;
```

Listing 12.2 Integration of the Router into the Application (src/App.tsx)

The `BrowserRouter` component enables the React Router functionality. Without this component, all other router components of the example can't work. The `Routes` component provides the frame for the individual routes defined under it. If the location of

the application changes, the `Routes` component searches all child elements for the appropriate path. In a `Routes` component, you may use only `Route` components or `React.Fragment` as direct child elements. For example, if you try to integrate an `h1` element, React Router acknowledges it with an exception.

In the example, you define three routes in the form of three `Route` components. Each of these components receives a `path` prop as well as an `element` prop. Using the `path` prop, you specify the path to which the route should respond, while the `element` prop must be used to specify which component should be rendered when the path is hit. So, for example, if you enable the `/list` path, the router makes sure that the `List` component is rendered.

The last route represents a special feature. Here you define the `/` path and render the `Navigate` component. The task of this route is to create a kind of bypass to the specified path. By default, you open your application with the `/` path, which causes the last route to be activated. The users are then redirected to the first defined route and the router renders the `List` component. For the code example to work, you still need to implement the `List` and `Form` components. For the first step, it's sufficient to keep the components simple. [Listing 12.3](#) contains the implementation of the `List` component and [Listing 12.4](#) the implementation of the `Form` component.

```
import React from 'react';

const List: React.FC = () => {
  return <h1>List works!</h1>;
};

export default List;
```

Listing 12.3 Basic Implementation of the “List” Component (src/List.tsx)

```
import React from 'react';

const Form: React.FC = () => {
  return <h1>Form works!</h1>;
};

export default Form;
```

Listing 12.4 Basic Implementation of the “Form” Component (src/Form.tsx)

In a case where no route applies, React Router renders nothing. So, in the worst case, the browser window remains empty for your users.

12.2.1 The Best Route Is Always Activated

You can never render a `Route` component directly. You must always wrap it in a `Routes` component. If you violate this rule, React Router throws a corresponding exception.

React Router ensures that exactly one route is rendered per `Routes` component. If two path specifications match for the current path in the application, the router chooses the more specific route. This may be the case, for example, if you define `Route` components with the `/books/new` and `/books/:id` paths. The second path contains a variable, which ensures that the route would be rendered for the `/books/1` path, but also on `/books/new`. In this case, however, the value of the `/books/new` path prop is more specific as it corresponds exactly to the specified path.

If you accidentally define the same route twice in a `Routes` component, React Router uses the first route it finds for a path and renders it.

However, this behavior doesn't mean that there is no way for you to react twice in different ways to a path in your application. To achieve this, you only need to render two `Routes` components.

With this state of the source code, you can navigate via the browser's address bar, but not yet within the application. To solve this problem, you add a navigation bar to your application.

12.2.2 A Navigation Bar for the Application

React Router only takes care of the navigation in the application, but not the styling of the components. For this reason, you use standard components of Material UI for the navigation bar. For the navigation bar, you use the `AppBar` component. Within this component, you use the `Toolbar` component to display the individual icons. A menu icon should be displayed on the left-hand side of the navigation bar. When you click on this icon, a navigation menu will appear, with the help of which you can switch to the list or the form.

To prevent the `App` component from becoming unnecessarily large, you swap out the navigation bar structure and associated logic to a separate component named `Nav`—namely, in the `Nav.tsx` file in the `src` directory. The source code of this component is shown in [Listing 12.5](#):

```
import React, { useState } from 'react';

import { AppBar, IconButton, Menu, MenuItem, Toolbar } from '@mui/material';
import { Menu as MenuIcon } from '@mui/icons-material';
import { Link } from 'react-router-dom';

const Nav: React.FC = () => {
  const [anchorEl, setAnchorEl] = useState<null | HTMLElement>(null);
```

```
function handleMenuOpen(event: React.MouseEvent<HTMLButtonElement>): void {
  setAnchorEl(event.currentTarget);
}

function handleMenuClose(): void {
  setAnchorEl(null);
}

return (
  <AppBar>
    <Toolbar sx={{ display: 'flex', justifyContent: 'space-between' }}>
      <IconButton
        edge="start"
        color="inherit"
        aria-label="Menu"
        onClick={handleMenuOpen}
      >
        <MenuIcon />
      </IconButton>
      <Menu
        id="navigation-menu"
        anchorEl={anchorEl}
        keepMounted
        open={Boolean(anchorEl)}
        onClose={handleMenuClose}
      >
        <MenuItem onClick={handleMenuClose}>
          <Link to="/list">Liste</Link>
        </MenuItem>
        <MenuItem onClick={handleMenuClose}>
          <Link to="/form">Formular</Link>
        </MenuItem>
      </Menu>
    </Toolbar>
  </AppBar>
);
};

export default Nav;
```

Listing 12.5 Navigation Bar for the Application (src/Nav.tsx)

The `handleMenuOpen` and `handleMenuClose` functions are responsible for opening and closing the navigation menu, respectively. For this purpose, the anchor element is set to a specific HTML element, in this case to the menu button or to the `null` value respectively.

Within the `AppBar` and `Toolbar` components, you first arrange an `IconButton` for the menu, the `Menu` component for the menu itself, and the **Logout** button. To the menu button, you bind the `handleMenuOpen` handler that opens the menu.

The menu itself is a component that defines the basic style properties and takes care of opening and closing the menu. The props of the menu are like those of a dialog, except that you can use the `anchorEl` prop to specify the anchor element responsible for positioning the menu. The `keepMounted` prop ensures that the child elements are kept in the DOM. This is relevant for the search engine optimization of an application, for example, as it allows the navigation structure to be found and processed by bots. Finally, inside the menu, you arrange the individual `MenuItem` components that represent the navigation options. In the example, the menu items contain `Link` components provided by React Router for the navigation.

For the styling of the navigation bar, you define the `sx` prop in the toolbar component, which ensures that the component uses the Flex layout and inserts space between the individual elements.

You won't see anything of the navigation bar in your application yet. For this reason, the next step is to take care of its integration.

12.2.3 Integrating the Navigation Bar

The simplest variant to display the navigation bar is to integrate it directly into the JSX structure of the `App` component. In [Listing 12.6](#), you can see how to implement the navigation bar integration:

```
import React from 'react';
import { BrowserRouter, Routes, Route, Navigate } from 'react-router-dom';
import List from './List';
import Form from './Form';
import Nav from './Nav';
import { Container } from '@mui/material';

const App: React.FC = () => {
  return (
    <BrowserRouter>
      <Nav />
      <Container sx={{ marginTop: '80px' }}>
        <Routes>
          <Route path="/list" element={<List />} />
          <Route path="/form" element={<Form />} />
          <Route path="/" element={<Navigate to="/list" />} />
        </Routes>
      </Container>
    </BrowserRouter>
  );
};
```



```
);
};

export default App;
```

Listing 12.6 Integration of the Navigation Bar in the “App” Component (src/App.tsx)

If you reload your application with this state of the code, you can log in and will then be redirected to its home page (see [Figure 12.1](#)). After logging in, you can use the navigation bar in any view of your application.

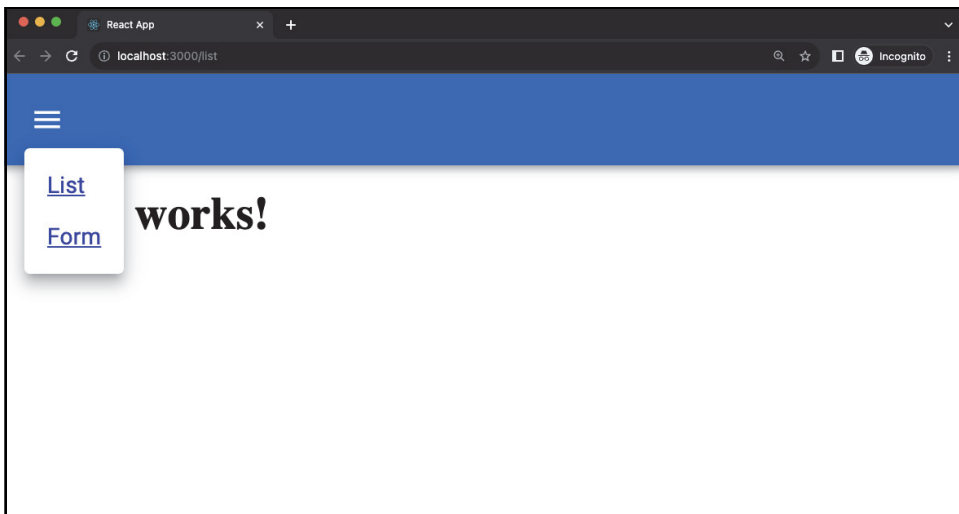


Figure 12.1 Navigation Bar in the Application

12.3 “NotFound” Component

As a result of the configuration, the manual input of an incorrect URL leads to the display of a white page in the browser. Such behavior is not very desirable, so you should make sure that a corresponding error page will be displayed. For this purpose, you can take advantage of the fact that only the first applicable route is rendered via the `Routes` component. You should therefore place another `Route` component in the `App` component with the path `*`, and render a component with an appropriate note. You can see how this works in detail in [Listing 12.7](#):

```
import React from 'react';
import { BrowserRouter, Routes, Route, Navigate } from 'react-router-dom';
import { Container } from '@mui/material';
import List from './List';
import Form from './Form';
import Nav from './Nav';
```

```
import NotFound from './NotFound';

const App: React.FC = () => {
  return (
    <BrowserRouter>
      <Nav />
      <Container sx={{ marginTop: '80px' }}>
        <Routes>
          <Route path="/list" element={<List />} />
          <Route path="/form" element={<Form />} />
          <Route path="/" element={<Navigate to="/list" />} />
          <Route path="*" element={<NotFound />} />
        </Routes>
      </Container>
    </BrowserRouter>
  );
};

export default App;
```

Listing 12.7 Default Route in the “App” Component (src/App.tsx)

For the change to work in the App component, you create the *NotFound.tsx* file in the *src* directory. The NotFound component informs users that an error has occurred and gives them the option to navigate to the home page—that is, the list view. The source code of the NotFound component is shown in [Listing 12.8](#):

```
import React from 'react';
import { Container } from '@mui/material';
import { Link } from 'react-router-dom';

const NotFound: React.FC = () => {
  return (
    <Container sx={{
      display: 'flex',
      flexDirection: 'column',
      alignItems: 'center'
    }}>
      <h1>Ups ... Something went wrong</h1>
      <p>
        <link to="/">Go to home page</link>
      </p>
    </Container>
  );
};
```

```
};
```

```
export default NotFound;
```

Listing 12.8 Implementation of the “NotFound” Component (src/NotFound.tsx)

To make both the heading and the text display centered, you render a container component from Material UI that you turn into a Flex container using the `sx` prop.

If you now enter the URL `http://localhost:3000/xxx` in the browser, the `NotFound` component will be rendered. The view should then look like the one shown in [Figure 12.2](#).

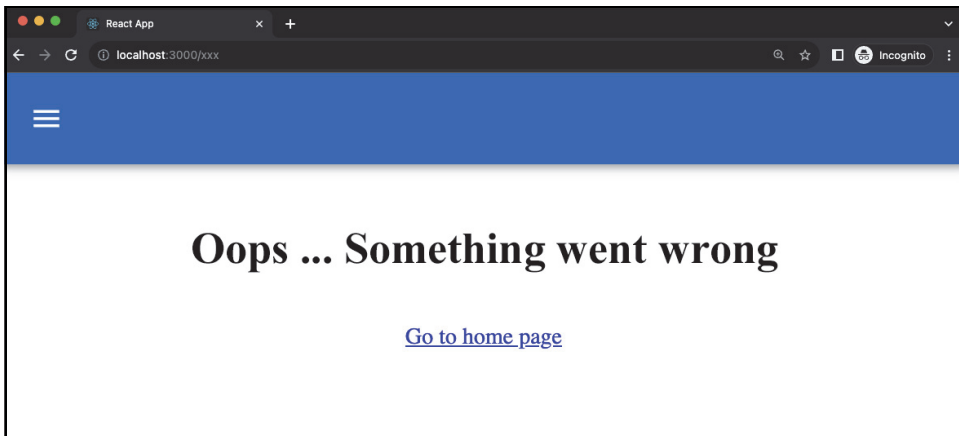


Figure 12.2 Display of the “NotFound” Component

Another exceptional case besides entering an incorrect URL is entering a URL that the user is not allowed to access because they lack permissions. You’ll see how this works in the remaining sections of this chapter.

12.4 Testing the Routing

Quick Start

The `MemoryRouter` enables you to test the routing process in your application. This component allows you to influence the current location of your application and control the routing for your test. For this purpose, you define the `initialEntries` prop in the `MemoryRouter` component:

```
<MemoryRouter initialEntries={['/list']}>...</MemoryRouter>
```

Listing 12.9 Influencing the Routing

The entry in [Listing 12.9](#) makes the application behave as if you had entered the path in the browser’s address bar.

When you use React Router in your application, you'll quickly find that manually testing the various options your application offers can quickly become laborious. For this reason, we recommend that you also secure the routes of your application by way of testing. Currently, the sample application has three routes, which you will test by means of unit tests in the following sections. The `App` component is a good place for such a test. In it, you can check if the application renders the correct components for a given route.

Before you get down to the real work, however, you should know that in addition to the `BrowserRouter` and `HashRouter` components, the React Router package contains a third router: the `MemoryRouter`. This is well suited for the integration into tests as it's lightweight and easy to influence. To be able to use the `MemoryRouter` in your `App` component, you still have to make a small modification and move the `BrowserRouter` from the `App` component to the `index.tsx` file. The adapted source code of the `index.tsx` file is shown in [Listing 12.10](#):

```
...
import { BrowserRouter } from 'react-router-dom';

const root = ReactDOM.createRoot(
  document.getElementById('root') as HTMLElement
);
root.render(
  <React.StrictMode>
    <BrowserRouter>
      <App />
    </BrowserRouter>
  </React.StrictMode>
);
...
```

Listing 12.10 Integration of “BrowserRouter” in the `index.tsx` File (`src/index.tsx`)

You can then remove `BrowserRouter` from your `App` component. Here, however, you must be careful to not simply remove the component, as your `App` component would then return more than just one root element. Instead of `BrowserRouter`, you return a React fragment as the root element. [Listing 12.11](#) shows the adjusted code of the `App` component:

```
const App: React.FC = () => {
  return (
    <>
      <Nav />
      <Container sx={{ marginTop: '80px' }}>
        <Routes>

```

```
    <Route path="/list" element={<List />} />
    <Route path="/form" element={<Form />} />
    <Route path="/" element={<Navigate to="/list" />} />
    <Route path="*" element={<NotFound />} />
  </Routes>
</Container>
</>
);
};
```

Listing 12.11 Adjustment of the “App” Component (src/App.tsx)

Having implemented these changes, you can now set about testing your application. A total of four tests are available. You can test the direct rendering of the `List` and `Form` components, then the redirect of the default route, and finally the entry of an invalid path. Because the four tests are very similar in structure and differ only in details, you can use the `describe.each` method of Jest to describe the test conditions in terms of an array of objects and have tests generated from them. [Listing 12.12](#) shows what this looks like:

```
import { render, screen } from '@testing-library/react';
import { MemoryRouter } from 'react-router-dom';
import App from './App';

const cases = [
  {
    description: 'should render the List',
    initialEntries: ['/list'],
    list: true,
    form: false,
    notFound: false,
  },
  {
    description: 'should render the Form',
    initialEntries: ['/form'],
    list: false,
    form: true,
    notFound: false,
  },
  {
    description: 'should provide a default route that points to /list',
    initialEntries: ['/'],
    list: true,
    form: false,
    notFound: false,
  },
];
```

```

    },
    {
      description: 'should render the NotFound component for invalid paths',
      initialEntries: ['/xxx'],
      list: false,
      form: false,
      notFound: true,
    },
  ],

  describe.each(cases)(
    'App',
    ({ description, initialEntries, list, form, notFound }) => {
      it(description, () => {
        render(
          <MemoryRouter initialEntries={initialEntries}>
            <App />
          </MemoryRouter>
        );

        const listExpect = expect(screen.queryByText('List works!'));
        const formExpect = expect(screen.queryByText('Form works!'));
        const notFoundExpect = expect(
          screen.queryByText('Ups ... Something went wrong')
        );

        list
          ? listExpect.toBeInTheDocument()
          : listExpect.not.toBeInTheDocument();
        form
          ? formExpect.toBeInTheDocument()
          : formExpect.not.toBeInTheDocument();
        notFound
          ? notFoundExpect.toBeInTheDocument()
          : notFoundExpect.not.toBeInTheDocument();
      });
    }
  );
};

```

Listing 12.12 Tests for the Routing of the Application (src/App.spec.tsx)

In the `cases` array, you define four objects, each of which contains the description of the test case in `description`. The `initialEntries` property indicates which path you want to test, and the `list`, `form`, and `notFound` properties indicate whether you expect the respective component to be visible.

You pass the `cases` array to the `describe.each` method. You also define the name of the test suite, which in this case is the name of the component: `App`.

This is followed by a callback function that Jest executes for each element of the `cases` array. Then you use a destructuring statement to extract the individual pieces of information and keep them in local variables. Using the `it` function, you then describe the generic test case, which you run a total of four times in the different variations. In doing so, you take the `App` component in the `MemoryRouter` during rendering and pass it the respective `initialEntries` to navigate to the corresponding path.

After this preparation, you locate the components according to the text displayed in each case. To make your tests more robust, you can also use test IDs at this point, but then you also need to insert them into the components. Note that you have to use the `queryBy*` functions of Jest here: `getBy*` functions such as `getByText` throw an exception if the element you are looking for can't be found. You can pass the three function calls directly to the `expect` function of Jest and then, depending on the test configuration, check whether the respective element should be included in the document or not.

When you run your tests with the code in this state using the `npm test` command, you'll get an output like that shown in [Figure 12.3](#).

```
PASS src/App.spec.tsx
App
  ✓ should render the List (57 ms)
  ✓ should render the Form (10 ms)
  ✓ should provide a default route that points to /list (9 ms)
  ✓ should render the NotFound component for invalid paths (10 ms)

Test Suites: 1 passed, 1 total
Tests:       4 passed, 4 total
Snapshots:  0 total
Time:        2.894 s, estimated 3 s
Ran all test suites related to changed files.

Watch Usage: Press w to show more.
```

Figure 12.3 Output of the Test Run

12.5 Conditional Redirects

There are situations when you want to render different components in your routes depending on the state of your application. A typical example involves applications that require you to log in. To demonstrate this, you first implement a simple `Login` component with one input field each for the user name and password. When you submit the form, the function passed to the component via the `onLogin` prop should be called with the entered user name and password. [Listing 12.13](#) contains the implementation of this component:

```

import React, { ChangeEvent, FormEvent, useState } from 'react';

type Props = {
  onLogin: (username: string, password: string) => void;
};

const Login: React.FC<Props> = ({ onLogin }) => {
  const [credentials, setCredentials] = useState({
    username: '',
    password: '',
  });

  function handleChange(event: ChangeEvent<HTMLInputElement>): void {
    setCredentials((prevCredentials) => ({
      ...prevCredentials,
      [event.target.name]: event.target.value,
    }));
  }

  function handleSubmit(event: FormEvent<HTMLFormElement>) {
    event.preventDefault();
    onLogin(credentials.username, credentials.password);
  }

  return (
    <form onSubmit={handleSubmit}>
      <div>
        <label htmlFor="username">Username: </label>
        <input
          type="text"
          value={credentials.username}
          onChange={handleChange}
          name="username"
          id="username"
        />
      </div>
      <div>
        <label htmlFor="password">Password: </label>
        <input
          type="password"
          value={credentials.password}
          onChange={handleChange}
          name="password"
          id="password"
        />
      </div>
    </form>
  );
};

```



```
        </div>
        <button type="submit">submit</button>
      </form>
    );
  };

export default Login;
```

Listing 12.13 Login Form (src/Login.tsx)

You implement the `Login` component using controlled components that are synchronized with the component's state. When submitting the form, you prevent the default behavior of the form by calling the `preventDefault` method and call the `onLogin` function with user name and password instead.

What's much more interesting than the `Login` component, which is not connected to React Router, is the implementation of the `App` component. In this component, you make sure that the `List` and `Form` components are rendered only if the users are logged in. You control the login via the local `handleLogin` function. Here you check if the user name *admin* and the password *test* have been entered. If so, you set the local `isLoggedIn` state to the value `true`. You also access this state when rendering the components in the element props of the `Route` components and display either the requested component or the login form. You can see the implementation of the `App` component with the changes for the login in [Listing 12.14](#):

```
import React, { useState } from 'react';
import { Routes, Route, Navigate, useNavigate } from 'react-router-dom';
import { Container } from '@mui/material';
import List from './List';
import Form from './Form';
import Nav from './Nav';
import NotFound from './NotFound';
import Login from './Login';

const App: React.FC = () => {
  const navigate = useNavigate();
  const [isLoggedIn, setLoggedIn] = useState(false);

  function handleLogin(username: string, password: string): void {
    if (username === 'admin' && password === 'test') {
      setLoggedIn(true);
      navigate('/');
    }
  }
}
```

```

return (
  <>
    <Nav />
    <Container sx={{ marginTop: '80px' }}>
      <Routes>
        <Route
          path="/list"
          element={isLoggedIn ? <List /> : <Navigate to="/login" />}
        />
        <Route
          path="/form"
          element={isLoggedIn ? <Form /> : <Navigate to="/login" />}
        />
        <Route path="/login" element={<Login onLogin= ↵
          {handleLogin} />} />
        <Route path="/" element={<Navigate to="/list" />} />
        <Route path="*" element={<NotFound />} />
      </Routes>
    </Container>
  </>
);
};

export default App;

```

Listing 12.14 Integration of the Local Login Process into the “App” Component

You implement the login on the client side only. For a real application, such an approach isn’t an option as the credentials are delivered in the code of the application—but for demonstration purposes, this variant is well suited as you have no dependencies on the server side.

The implementation of the login consists of the Boolean `isLoggedIn` state of the `App` component and the `handleLogin` function. You pass this function to the `Login` component, which in turn calls it with the entered user name and password. The `handleLogin` function checks whether the user name, here the `admin` string, and the password, `test`, have been entered. If that’s the case, you set the `loggedIn` state to `true` and use the `navigate` function of `React Router` to redirect users to the default route. You can get the `navigate` function via the `useNavigate` hook of `React Router`. This function is the program-based counterpart to the declarative `Navigate` component.

In the `/list` and `/form` routes, you check within the `element` prop to see if users are currently logged in. If so, you render the `List` or `Form` component. Otherwise, you redirect to the `/login` route.

12.6 Dynamic Routes

In the next step, we'll take care of using the form as a dialog in the list view. In this context, you'll learn about variables and subroutes, which are two important features of React Router. To be able to edit records, you need to be able to pass variables to a route to specify which data record should be edited. You define two separate routes for editing and creating data records. It should be possible to edit the data record with ID 2 via `/list/edit/2` and to create a new data record via `/list/new`. The dialog management should be done completely through the router, so you don't need an additional state to manage the dialog.

12.6.1 Defining Subroutes

The `Route` component of React Router enables you to define nested routes. This allows you to determine whether additional components are rendered at specific locations in a component tree depending on the currently selected URL path. It also means that you can implement new routes for dialog handling within the `/list` route, for example. In [Listing 12.15](#), you can see how these two routes are integrated into the `App` component:

```
import React from 'react';
import { Routes, Route, Navigate } from 'react-router-dom';
import List from './List';
import Form from './Form';
import NotFound from './NotFound';

const App: React.FC = () => {
  return (
    <Routes>
      <Route path="/list" element={<List />}>
        <Route path="edit/:id" element={<Form />} />
        <Route path="new" element={<Form />} />
      </Route>
      <Route path="/" element={<Navigate to="/list" />} />
      <Route path="*" element={<NotFound />} />
    </Routes>
  );
};

export default App;
```

Listing 12.15 Integration of Subroutes into the “App” Component (src/App.tsx)

At the highest level, the `App` component contains three routes: the `/list` route to display the list, the `/` route as the default route that redirects to the list display, and the `NotFound` route if an invalid path has been specified.

Below the `/list` route, you define two subroutes: `edit/:id` and `new`. Note here that the two routes don't start with `/` because they are subroutes of the absolute route, `/list`. This configuration now enables you to control the list as well as the form dialog.

Implementation of the “List” Component

The actual logic is contained in the `List` and `Form` components, respectively. [Listing 12.16](#) shows the implementation of the `List` component:

```
import React, { useEffect, useState } from 'react';
import {
  IconButton, List as MuiList, ListItem, ListItemText, Fab,
} from '@mui/material';
import { Book } from './Book';
import { Link, Outlet, useLocation } from 'react-router-dom';
import { Add, Edit } from '@mui/icons-material';

const List: React.FC = () => {
  const location = useLocation();
  const [books, setBooks] = useState<Book[]>([]);
  useEffect(() => {
    if (location.pathname === '/list') {
      fetch('http://localhost:3001/books')
        .then((response) => response.json())
        .then((data) => setBooks(data));
    }
  }, [location]);

  return (
    <>
      <MuiList>
        {books.map((book) => (
          <ListItem key={book.id}>
            <ListItemText>{book.title}</ListItemText>
            <IconButton
              aria-label="edit"
              component={Link}
              to={` /list/edit/${book.id}`}
            >
              <Edit />
            </IconButton>
          </ListItem>
        ))}
      </MuiList>
      <Fab color="primary" aria-label="Add" component={Link} to="/list/new">
        <Add />
      </Fab>
    </>
  );
};
```

```
        </Fab>
        <Outlet></Outlet>
      </>
    );
  };

  export default List;
```

Listing 12.16 Display of Data as a List (src/List.tsx)

The `List` component displays the data records as a Material UI list, which, to avoid naming conflicts, you rename to `MuiList` during the import process. The `List` component has its own state in which you store the data records of type `Book`. Initially the state consists of an empty array. The `useLocation` hook of React Router allows you to access the current location and thus also the path name. Depending on this path name, you execute an effect hook. If the pathname is `/list`, you load the data. This ensures that each time the list is displayed, the data is reloaded from the server. This is true both when the component is initially rendered and when the form dialog is closed, so in each case the list updates to show the current state of the data.

For the list display, you use the `List` component of Material UI, and its subcomponents `ListItem` and `ListItemText` to display the title of the books. In each row, you also render an `IconButton` that contains the **Edit** icon. The `IconButton` of Material UI allows you to define a component for the button using the `component` prop. In this case, you use the `Link` component of React Router. The `to` prop is used to specify that the `/list/edit/<id>` route should be activated when the button is clicked on, where `<id>` stands for the respective data ID of the record. You can proceed in a similar way with the fab for new records.

If you now click on one of the **Edit** buttons or the fab, React Router will activate the respective subroute, which results in the `/list` route being activated first and the `List` component being rendered. Also, the router renders the subroute, either `new` or `edit/:id`, and renders the component specified there in the `App` component into the `Outlet` component of the `List` component. As a result, the form dialog is displayed to users above the list.

Implementation of the Form Dialog

The `Form` component, which contains the dialog control and the actual form, is a bit more complex to implement because it also takes care of the communication with the server. The form is based on React Hook Form, which means you don't need to bother with form handling yourself. You can see the source code of the `Form` component in [Listing 12.17](#):

```
import {
  Button,
  Dialog,
```

```

    DialogActions,
    DialogContent,
    DialogTitle,
  } from '@mui/material';
import React, { useEffect } from 'react';
import { useForm } from 'react-hook-form';
import { useNavigate, useParams } from 'react-router-dom';
import { InputBook } from './Book';

const Form: React.FC = () => {
  const { register, handleSubmit, reset } = useForm<InputBook>({
    defaultValues: {
      title: '',
      author: '',
      isbn: '',
    },
  });
  const navigate = useNavigate();
  const { id } = useParams<{ id: string }>();

  useEffect(() => {
    if (id) {
      fetch(`http://localhost:3001/books/${id}`)
        .then((response) => response.json())
        .then((data) => reset(data));
    }
  }, [id, reset]);

  async function handleSave(formData: InputBook) {
    let url = 'http://localhost:3001/books';
    let method = 'POST';

    if (formData.id) {
      method = 'PUT';
      url += `/${formData.id}`;
    }

    await fetch(url, {
      method,
      body: JSON.stringify(formData),
      headers: { 'content-type': 'application/json' },
    });
    handleClose();
  }
}

```

```
function handleClose() {
  navigate('/list');
}

return (
  <Dialog
    open={true}
    onClose={handleClose}
    aria-labelledby="form-dialog-title"
    aria-describedby="form-dialog-description"
  >
    <form onSubmit={handleSubmit(handleSave)}>
      <DialogTitle id="form-dialog-title">
        {id ? 'Edit book' : 'Create new book'}
      </DialogTitle>
      <DialogContent id="form-dialog-description">
        <div>
          Title:
          <input {...register('title')} />
        </div>
        <div>
          Author:
          <input {...register('author')} />
        </div>
        <div>
          ISBN:
          <input {...register('isbn')} />
        </div>
      </DialogContent>
      <DialogActions>
        <Button color="secondary" onClick={handleClose}>
          Cancel
        </Button>
        <Button color="primary" type="submit">
          Saving
        </Button>
      </DialogActions>
    </form>
  </Dialog>
);
};

export default Form;
```

Listing 12.17 Implementation of the Form Dialog (src/Form.tsx)

In the `Form` component, you first use the `useNavigate` hook to create the `navigate` function, which you can use to switch back to the list view and close the dialog. You can access the variables in the URL path via the `useParams` hook. In this case, that's the `id` variable. The `useParams` function is defined as a generic function, so you can specify the expected structure here. When you call the `useForm` hook, you define an empty data record as the default value that you initially load into the form. So in the case of a new data record, all fields are empty. The `effect` hook, which depends on the `id` property from the URL path, causes the data to be loaded from the server and written to the form using the `reset` function of `React Hook Form` when you want to edit a record.

The `handleSave` function is responsible for saving a data record, covering both new and existing records. Once the save operation has been completed, this function calls the `handleClose` function, which navigates back to the `/list` route and thus closes the dialog.

The JSX structure of the component consists of a `Dialog` component whose `open` prop always has the value `true`, so the dialog is always open when it is rendered. You pass the `handleClose` function to the `onClose` prop so that you can close the dialog via the router. The form itself is a simple `React Hook Form` form with fields for title, author, and ISBN. For the implementation to work, you must make sure that the `react-hook-form` package is installed. You also need a backend that provides the book records. Here you can draw on `json-server` from the previous chapters. Once you've ensured that both the backend and frontend processes are running, you can switch to the browser and should see a view like the one shown in [Figure 12.4](#) when you edit a data record.

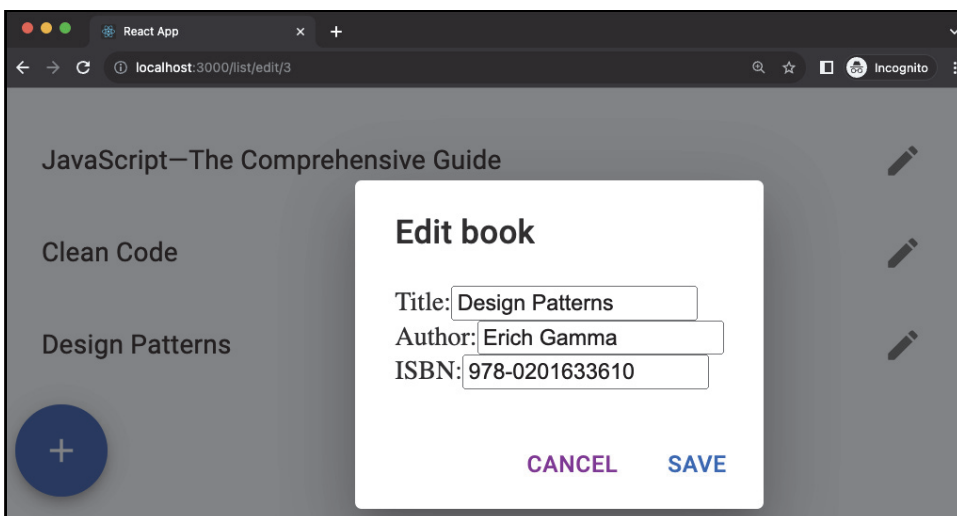


Figure 12.4 Editing Data Records via Subroutes

12.7 Summary

This chapter addressed the topic of navigating in a single-page application. For a convenient management of this feature, React Router has been introduced:

- You now know how to install the router and integrate it with your application.
- You also know the difference between the HTML5 History API and hash navigation, and you know that React Router supports both variants.
- React Router works declaratively with the `Routes` component as a container and `Route` to define individual routes.
- The router always activates the best-matching route based on the current path as well as on the `path` prop of the `Route` component.
- The `Link` component allows navigation from within a JSX structure.
- The `path` value of `*` allows you to define a wildcard route to respond to invalid paths.
- When testing applications with routes, you use `MemoryRouter`. It's lightweight, and you can control it in your test environment.
- By rendering the `Navigate` component, you can redirect a user to another route.
- The `useParams` hook enables you to access the variables of a route.
- You can use the `useLocation` hook of the router to access various aspects of the current route, such as the URL path.
- The `useNavigate` hook enables you to activate a different route from within the component logic.
- Subroutes allow you to render subordinate components.
- The router also allows you to manage Material UI dialogs and to open and close them via the URL.

In the next chapter, you'll learn how to create your own libraries and thus use components and other structures in multiple applications.

Chapter 13

Creating Custom React Libraries

Often, the applications you develop using React stand on their own. Then you implement components and other structures that you use in a self-contained environment over which you have almost complete control. However, there are also situations where you need to use your components in multiple applications. This is especially the case if you are in a larger enterprise context with multiple applications. In this case, it makes little sense for you to reimplement the standard components used for all the company's applications, thus reinventing the wheel over and over again.

In this chapter, we'll look at how you can create your own component library to use components and hook functions in more than just a single application.

13.1 Creating a Custom Component Library

Quick Start

To create your own component library, you need to perform several steps:

1. You need to initialize your library.
2. Then you configure the build system.
3. You implement the components of the library.
4. In the final step, you determine how to make the library available.

When using React, sooner or later you'll integrate libraries to extend its functionality. You use React Router to navigate your application, Redux for centralized state management, or Material UI to access a collection of prebuilt components. Libraries for React all have the same goal: they create reusable structures to solve standard problems at a higher level than just that of a single application.

If you work in an organization where multiple React applications are being developed, you'll quickly find that all development teams use a basic set of very similar components. However, you can avoid this multiple work by creating and developing these components in a central location. You can then integrate this central library into your applications via your package manager, such as NPM, for example, and use it there.

The library we create in this chapter provides a component as well as a hook function. It's implemented based on TypeScript, the components have their own styles and tests, and you use Storybook to display the components before they're integrated.

13.1.1 Initializing the Library

The basis for the library is a new directory named *library*. In this directory, you first create a `.gitignore` file with the content from [Listing 13.1](#) to prevent `node_modules` from being integrated into your repository:

```
node_modules
```

Listing 13.1 Excluding the `node_modules` Directory from Version Control (`library/.gitignore`)

In the next step, you use the `npm init -y` command to create a `package.json` file for your library. The command creates a default version of the description file for your library, which you can adjust somewhat, as you can see in [Listing 13.2](#):

```
{
  "name": "library",
  "version": "1.0.0",
  "description": "Utility library",
  "scripts": {},
  "license": "ISC"
}
```

Listing 13.2 Description File for the Library (`library/package.json`)

At this point, the only important thing is the name of the library, which you specify here as `library`.

So far in the examples you have used Create React App, which in turn uses Webpack as a bundler. However, libraries often use Rollup as a bundler. One reason for this is that this tool is much more lightweight to configure than Webpack. For the build process of your library, you must first install several packages. To do this, you need to enter the command in [Listing 13.3](#):

```
npm install rollup \n
  @rollup/plugin-commonjs \n
  @rollup/plugin-node-resolve \n
  @rollup/plugin-typescript \n
  rollup-plugin-dts \n
  rollup-plugin-peer-deps-external \n
  rollup-plugin-terser
```

Listing 13.3 Installing the Dependencies for the Build Process

Table 13.1 provides a brief explanation of each package you have just installed.

Package name	Meaning
rollup	This package includes the Rollup bundler itself.
@rollup/plugin-commonjs	This plugin converts CommonJS modules to ECMAScript modules.
@rollup/plugin-node-resolve	This implements the Node.js package resolution for Rollup to include third-party modules.
@rollup/plugin-typescript	This provides TypeScript support for Rollup.
rollup-plugin-dts	This plugin aggregates the TypeScript definition files.
rollup-plugin-peer-deps-external	This plugin ensures that the peer dependencies of the library are not included in the bundle.
rollup-plugin-terser	This plugin allows you to optimize the package size of your library.

Table 13.1 Dependencies for the Build Process

After installing these packages, the next step is to configure Rollup. You save this configuration in a file named *rollup.config.js*. You can see the contents of this file for the library in [Listing 13.4](#):

```
import commonjs from '@rollup/plugin-commonjs';
import resolve from '@rollup/plugin-node-resolve';
import typescript from '@rollup/plugin-typescript';
import dts from 'rollup-plugin-dts';
import peerDepsExternal from 'rollup-plugin-peer-deps-external';
import { terser } from 'rollup-plugin-terser';

const config = [
  {
    input: 'src/index.ts',
    output: [
      {
        file: 'dist/cjs/index.js',
        format: 'cjs',
        sourcemap: true,
      },
    ],
  },
  {
    file: 'dist/esm/index.js',
    format: 'esm',
    sourcemap: true,
  },
];
```

```
    },
  ],
  plugins: [
    resolve(),
    commonjs(),
    typescript({ tsconfig: './tsconfig.json' }),
    peerDepsExternal(),
    terser(),
  ],
},
{
  input: 'dist/esm/types/index.d.ts',
  output: [{ file: 'dist/index.d.ts', format: 'es' }],
  plugins: [dts()],
},
];
```

```
export default config;
```

Listing 13.4 Rollup Configuration (library/rollup.config.js)

The Rollup config array contains two objects. The first object is responsible for bundling the library itself, and the second object takes care of generating the type definitions. You use the `input` property to specify the entry point to your library. For the example, this is the `index.ts` file in the `src` directory. As output, you define an array with two objects. One of them represents the library in CommonJS format, while the other represents the ECMAScript module system. In both cases, you have source maps generated for better debugging.

Finally, you use the `plugin` property to define which plugins Rollup should apply to your library's source code. On the one hand, these are the `resolve` and `commonjs` functions, which come from the `@rollup/plugin-node-resolve` and `@rollup/plugin-commonjs` packages, and on the other hand, you use the TypeScript plugin so that you can implement your library in TypeScript. The last two plugins, `peerDepsExternal` and `terser`, ensure that the bundle size is optimized.

With the TypeScript configuration, you're still missing an important building block in the setup of your library. You can generate this file using the `npx tsc --init` command. You'll then need to modify the default file to suit the requirements of your library. The result is shown in [Listing 13.5](#).

```
{
  "compilerOptions": {
    "target": "es2016",
    "esModuleInterop": true,
    "forceConsistentCasingInFileNames": true,
```

```

    "strict": true,
    "skipLibCheck": true,
    "module": "ESNext",
    "jsx": "react-jsx",
    "declaration": true,
    "declarationDir": "types",
    "sourceMap": true,
    "outDir": "dist",
    "moduleResolution": "node",
    "emitDeclarationOnly": true
  }
}

```

Listing 13.5 TypeScript Configuration (library/tsconfig.json)

13.1.2 The Structure of the Library

Like a regular application, React doesn't give you any guidance on the structure in the file system when building a library. You should organize your components in a directory structure that ensures that developers can quickly find their way around and easily locate the individual components of the library.

For this purpose, you first create a directory named *src* in the root directory of your application, where you store the elements separately from the library configuration. Then you can differentiate by the type of structure and create directories such as *components* and *hooks*.

Inside the *components* directory, you create another directory named *Button*, where you implement the first component of your library. You save the source code of the Button component in a file named *Button.tsx*. The corresponding source code is shown in [Listing 13.6](#):

```

import React, { MouseEvent } from 'react';
import { StyledButton } from './Button.style';

type Props = {
  children: string;
  onClick: (event: MouseEvent<HTMLButtonElement>) => void;
};

const Button: React.FC<Props> = ({ children, onClick }) => {
  return <StyledButton onClick={onClick}>{children}</StyledButton>;
};

export default Button;

```

Listing 13.6 Implementation of the Button Component (library/src/components/Button.tsx)

For your `Button` component, you first define the structure of the props that are expected by the component. Specifically, these are the `children` and the `onClick` props. The `children` prop stands for the label of the button. You can pass this label either via the explicit `children` prop or you pass the contents between the opening and closing component tags. The `onClick` prop receives the click handler, which is the function that's executed when the button is interacted with.

The implementation of the button consists of the `StyledButton` component, to which you apply the two props you specified previously. To style the `Button` component, you use `Emotion` or the `@emotion/styled` package to create the `StyledButton` component, whose source code you store in the `Button.style.tsx` file in the `Button` directory:

```
import styled from '@emotion/styled';

export const StyledButton = styled.button`
  padding: 5px;
  border-radius: 5px;
  background-color: lightgray;
  &:hover {
    background-color: white;
  }
`;
```

Listing 13.7 Styling of the “Button” Component (library/src/components/Button/Button.style.tsx)

By creating the `StyledButton` in [Listing 13.7](#), you've created the basic structure of the component. Then you assign multiple styles to the `StyledButton` component and define a hover state.

You can become as generic as you like, both in the button implementation itself and in the styling. With component libraries, you typically define generic components, whose appearance and behavior you can control using props, and which you can use in multiple places in your application or in multiple applications. The `Button` component in this example is deliberately kept simple, as the focus here is on creating the library itself and integrating the components into an application.

To be able to use the components of your library comfortably, you export the individual components of the library up to the central `index.ts` file. This has the advantage for users that they don't need to know the internal structure of the library. For this purpose, you create one `index.ts` file per subdirectory that takes care of the export. The `index.ts` file in the `Button` directory is the first one:

```
export { default } from './Button';
```

Listing 13.8 Export to the “Button” Directory (library/src/components/Button/index.tsx)

The next level is then the *index.ts* file in the *components* directory. It collects the exports of all component directories. So if you implement additional components, you must add more entries in this file for the respective components:

```
export { default as Button } from './Button';
```

Listing 13.9 Export to the components Directory (library/src/components/index.ts)

The final step consists of implementing the *index.ts* file in the *src* directory. This file exports all the structures of your library to a central location, which ensures that you don't need to memorize paths when integrating, but can import the structures directly. This file is thus the entry point into your application.

```
export * from './components';
```

Listing 13.10 Central Export in the src Directory (library/src/index.ts)

In such a library, you can collect not only components, but also other structures like hooks or general helper functions that you can use in your application.

13.1.3 Hooks in the Library

One example of a hook you can implement in your library is a function that takes care of loading data. This hook function is named *useLoadData* and is located in a file of the same name in the *src/hooks/useLoadData* directory. The implementation of this hook function is shown in [Listing 13.11](#):

```
import { useState, useEffect } from 'react';

function useLoadData<T>(url: string): T[] {
  const [state, setState] = useState<T[]>([]);

  useEffect(() => {
    fetch(url)
      .then((response) => response.json())
      .then((data) => setState(data));
  }, []);

  return state;
}

export default useLoadData;
```

Listing 13.11 Implementation of the “useLoadData” Hook (library/src/hooks/useLoadData/useLoadData.ts)

You implement the `useLoadData` function as a generic function in TypeScript. This means that when you call it, you can determine the types of the objects you load from the server. As a parameter, you pass the URL of the endpoint from which the data will be loaded. The return value is an array of objects of the generic type.

As with the component, you must export the hook function up to the `index.ts` file in the `src` directory. It starts with the `index.ts` file in the `src/hooks/useLoadData` directory:

```
export { default } from './useLoadData';
```

Listing 13.12 Export to the `useLoadData` Directory (`library/src/hooks/useLoadData/index.ts`)

In the next step, you export the hook function in the `hooks` directory, as shown in [Listing 13.13](#):

```
export { default as useLoadData } from './useLoadData';
```

Listing 13.13 Export in the `hooks` Directory (`library/src/hooks/index.ts`)

In the final step, you export the hook function in the `index.ts` file in the `src` directory:

```
export * from './components';  
export * from './hooks';
```

Listing 13.14 Export in the Entry File (`library/src/index.ts`)

At this point, you've prepared the library to the point where you can build it.

13.1.4 Building the Library

When you initialize your library, you've already done most of the preparation for building the library. When you run the Rollup bundler with the current configuration, your library gets built in the `dist` directory and you can start using it. You can simplify the process a little more by adding a new script to your `package.json` file. Before editing the file, you need to use the `npm install rimraf` command to install an additional package that allows you to delete files and directories. The updated source code of the `package.json` file is shown in [Listing 13.15](#):

```
{  
  "name": "library",  
  "version": "1.0.0",  
  "description": "Utility library",  
  "scripts": {  
    "prebuild": "rimraf dist",  
    "build": "rollup -c"  
  },  
  "license": "ISC",  
  "dependencies": {
```

```

    "@rollup/plugin-commonjs": "^22.0.0",
    "@rollup/plugin-node-resolve": "^13.3.0",
    "@rollup/plugin-typescript": "^8.3.2",
    "rimraf": "^3.0.2",
    "rollup": "^2.74.1",
    "rollup-plugin-dts": "^4.2.2",
    "rollup-plugin-peer-deps-external": "^2.2.4",
    "rollup-plugin-terser": "^7.0.2",
    "@types/react": "^18.0.9",
  },
  "peerDependencies": {
    "@emotion/react": "^11.9.0",
    "@emotion/styled": "^11.8.1"
    "react": "^18.1.0",
  },
  "main": "dist/esm/index.js",
  "types": "dist/index.d.ts"
}

```

Listing 13.15 package.json Script for the Build Process (library/package.json)

To ensure that the build process runs without errors, you need to install the type definitions for React and Emotion via the `npm install @types/react @emotion/react @emotion/styled` command. In addition to these dependencies, you can also add React as a peer dependency (with `peerDependency`) in the *package.json* file to indicate that your library cannot work without installing React in the application. Furthermore, you can also specify the Emotion packages with `peerDependency` to make sure that they are also installed by the application and not indirectly by the library.

In a final step, you specify the entry point to your application with the `main` field and the reference to the type definitions with the `types` field.

At this state of your library, you can build it on the command line. For this purpose, you need to go to the root directory of the library and call the `npm run build` command. The result should be two success messages. Rollup reports that it has created the *dist/cjs/index.js* and *dist/esm/index.js* files on the one hand, and the type definitions under *dist/index.d.js* on the other.

13.2 Integrating the Library

You can now use the application you just built in several ways in an application. Depending on the requirements you have, different approaches are possible. What they all have in common is that you first need to set up a working React application. You can do this using the `npx create-react-app books --template typescript` command. Run this command in the directory where you also created your library.

Your library meets all the requirements for a valid NPM package, so you can install it using a package manager such as NPM or Yarn. Several options are available here. Especially when you are actively developing your library, you can use the `npm link` command to integrate the current development state of the library into your application. Changes to the library will then take effect without an additional reinstallation or update.

When you switch to your application, you can run the `npm link ../library` command. This causes NPM to create a symbolic link to your library in the `node_modules` directory. Here, however, you must note that your library must not have the `react` package installed. For this reason, you need to remove the `react`, `@emotion/styled`, and `@emotion/react` packages there. Because you've defined the `@emotion` packages as `peerDependencies`, you need to install them in your application using the `npm install @emotion/react @emotion/styled` command. Then the integration of your library will work. Because the `useLoadData` hook needs a backend to work, you also install the `json-server` package via the `npm install json-server` command. To run the backend, you need a JSON file, which you store in the root directory of the application and name *data.json*. The structure of this file is shown in [Listing 13.16](#):

```
{
  "books": [
    {
      "id": 1,
      "title": "JavaScript - The Comprehensive Guide",
      "author": "Philip Ackermann",
      "isbn": "978-3836286299",
      "rating": 5
    },
    {
      "id": 2,
      "title": "Clean Code",
      "author": "Robert Martin",
      "isbn": "978-0132350884",
      "rating": 4
    },
    {
      "id": 3,
      "title": "Design Patterns",
      "author": "Erich Gamma",
      "isbn": "978-0201633610",
      "rating": 5
    }
  ]
}
```

Listing 13.16 Data for the Server (books/data.json)

You can conveniently start the server by adding the line "backend": "json-server -p 3001 -w data.json" in the scripts section of your *package.json* file and then running the `npm run backend` command. After that, you start the frontend in a second command line using the `npm start` command. You can then customize the App component of your new application and include both the `useLoadData` hook and the `Button` component. You can see how this works in [Listing 13.17](#):

```
import React from 'react';
import './App.css';
import { Book } from './Book';
import { useLoadData, Button } from 'library';

const App: React.FC = () => {
  const books = useLoadData<Book>('http://localhost:3001/books');

  return (
    <>
      <ul>
        {books.map((book) => (
          <li key={book.id}>{book.title}</li>
        ))}
      </ul>
      <Button onClick={() => console.log('Button clicked')}> ←
        click me</Button>
    </>
  );
}

export default App;
```

Listing 13.17 Integration of the Library into the Application (books/src/App.tsx)

First, you import the `useLoadData` function and the `Button` component as a named import from the `library` package. Then you call the hook function with the backend URL and get the data records that the hook function stores in its local state. This is initially an empty array, and after successful server communication it's an array with three data records. The state update causes React to re-render the App component and display the records correctly.

Then you integrate the `Button` component. After that, you pass a label and register a callback function as a click handler that generates output to the console. When you check your application in the browser now, you should see a view like that in [Figure 13.1](#).

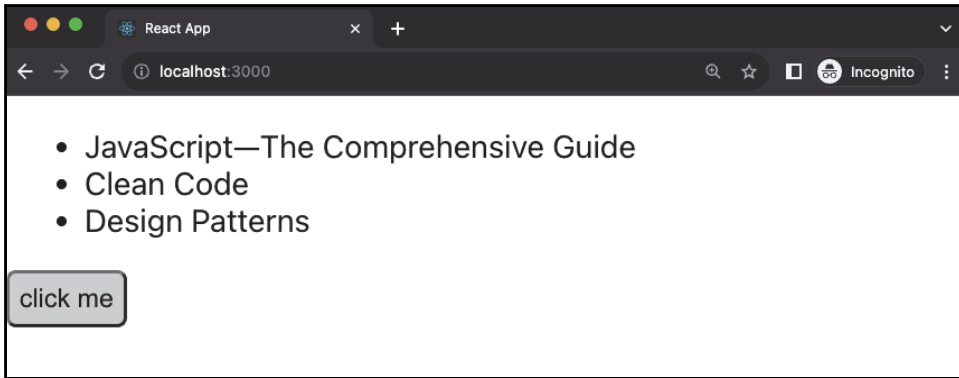


Figure 13.1 Using the Library Structures in the Application

13.2.1 Regular Installation of the Package

The `npm link` command is just one of several ways you can integrate your library into an application. The disadvantage of this method is that while it is well suited for development use, it isn't an option for production use. Instead, you can choose from the following variants:

■ Local installation

You can use NPM also to specify a local package as a directory or a zipped TAR file instead of a regular package that resides in the NPM registry. The minimum requirement is that the specified structure contains a *package.json* file. The disadvantage of this variant is that the package must be available locally on each computer; otherwise the installation of the application will fail.

■ Installation from a server

Instead of specifying a local source, you can also specify a server where the package is located as a zipped TAR file.

■ Git repository

Another option is to reference a Git repository that contains your package during installation.

■ Publishing the package

As an alternative to the previous options, you can also publish your package and then install it like an ordinary NPM package. For a public package that can be accessed by all NPM users, you only need a free NPM account. Private packages are a service provided by NPM, but you need to pay for them. In this case, only authorized persons can access the package. By using packages like Verdaccio, you can also build your own NPM registry and publish your packages there.

For more information on publishing packages, visit <https://docs.npmjs.com/packages-and-modules/contributing-packages-to-the-registry>.

If you intend to use your library in your application or make it available to others, you need to take care of the automated validation of your library's features through testing.

13.3 Testing the Library

If you implement a library, you should expect at least the same quality level from it as from a regular application, if not even a higher one. After all, a library tends to be used in more than one application, and so a bug would have a wider reach. One of the most obvious means by which you can secure your library is to use automated unit tests. You can formulate these tests using Jest, as in your application.

13.3.1 Preparing the Testing Environment

Before you start writing your unit tests, you need to prepare the environment and install several packages for it. To do this, you want to run the following command:

```
npm install --save-dev @testing-library/react jest @types/jest babel-
jest @babel/preset-env @babel/preset-react @babel/preset-typescript
```

This will install Jest, the Babel compiler, and several plugins for the compiler. Then you have Jest create a *jest.config.js* file for you using the `npx jest --init` command. This command starts an interactive mode that guides you through a series of questions. For each of these, you can confirm the default response with `[Enter]`, with the exception of the testing environment: here you select the `jsdom` (browser-like) option. The final step is to create a configuration for Babel in the *babel.config.js* file in the root directory of your library. The source code of this file is shown in [Listing 13.18](#):

```
module.exports = {
  presets: [
    '@babel/preset-env',
    ['@babel/preset-react', { runtime: 'automatic' }],
    '@babel/preset-typescript',
  ],
};
```

Listing 13.18 Babel Configuration for the Unit Tests (library/babel.config.js)

Now all you need are the actual unit tests, which you'll write in the following sections.

13.3.2 Unit Test for the Library Component

To test the `Button` component of the library, you create the *Button.spec.tsx* file in the *src/components/Button* directory in your library. In the test, you render the button,

click on it, and expect the callback function passed in the `onClick` prop to be executed. The corresponding source code is shown in [Listing 13.19](#):

```
import { fireEvent, render, screen } from '@testing-library/react';
import Button from './Button';

describe('Button', () => {
  it('should call the onClick-Function when clicked', () => {
    const onClick = jest.fn();
    render(<Button onClick={onClick}>click me</Button>);
    const button = screen.getByText('click me');
    fireEvent.click(button);
    expect(onClick).toHaveBeenCalled();
  });
});
```

Listing 13.19 Test of the “Button” Component (library/src/components/Button/Button.spec.tsx)

Because Jest creates a test script in the *package.json* file by default when initializing the configuration, you can run the test directly using the `npm test` command.

13.3.3 Unit Test for the Custom Hook of the Library

Because the library's custom hook communicates with a server, you need a tool to mock the server communication. In this case, you run the `npm install --save-dev msw whatwg-fetch @testing-library/jest-dom` command to install the Mock Service Worker package, a fetch polyfill, and the `jest-dom` extension of the Testing Library. The polyfill is required because Jest runs in Node.js and the Fetch API isn't currently available there. The extension provides you with some helper functionality—for example, the `toHaveTextContent` matcher, which you can use to make your tests more convenient.

After the installation, you create a new file named *useLoadData.spec.tsx* in the *src/hooks/useLoadData* directory. There, you first configure Mock Service Worker and create a test component that uses the `useLoadData` hook and renders a simple list of data from the server. In the final step, you formulate the actual test, which renders the test component and then checks whether the data is output correctly by the server. The source code of the test is shown in [Listing 13.20](#):

```
import 'whatwg-fetch';
import { rest } from 'msw';
import { setupServer } from 'msw/node';
import useLoadData from './useLoadData';
import { ReactElement } from 'react';
import { render, screen } from '@testing-library/react';
```

```
import '@testing-library/jest-dom';

const books = [
  {
    id: 1,
    title: 'JavaScript - The Comprehensive Guide',
    author: 'Philip Ackermann',
    isbn: '978-3836286299',
    rating: 5,
  },
  {
    id: 2,
    title: 'Clean Code',
    author: 'Robert Martin',
    isbn: '978-0132350884',
    rating: 2
  },
  {
    id: 3
    title: 'Design Patterns',
    author: 'Erich Gamma',
    isbn: '978-0201633610',
    rating: 5,
  },
];

const server = setupServer(
  rest.get('/books', (req, res, ctx) => {
    return res(ctx.json(books));
  })
);

function TestComponent(): ReactElement {
  const books = useLoadData<{ id: number; title: string }>('/books');
  return (
    <ul>
      {books.map((book) => (
        <li key={book.id} data-testid="item">
          {book.title}
        </li>
      ))}
    </ul>
  );
}
```



```
describe('useLoadData', () => {
  beforeAll(() => server.listen());
  afterEach(() => server.resetHandlers());
  afterAll(() => server.close());

  it('should fetch and return data correctly', async () => {
    render(<TestComponent />);
    const books = await screen.findAllByTestId('item');
    expect(books).toHaveLength(3);
    expect(books[0]).toHaveTextContent('JavaScript - ⬅️  
The Comprehensive Guide');
    expect(books[1]).toHaveTextContent('Clean Code');
    expect(books[2]).toHaveTextContent('Design Patterns');
  });
});
```

Listing 13.20 Test of the “useLoadData” Function
(library/src/hooks/useLoadData/useLoadData.spec.tsx)

You can also run this test in your library using the `npm test` command, and you should receive a positive result in the command line.

Besides testing, there’s another aspect that is especially interesting for those who include a library: What do the components actually look like, and how can they be configured? This is best demonstrated with tools such as Storybook.

13.4 Storybook

Storybook is a tool that allows you to develop your components independently of a React application. The graphical user interface allows you to demonstrate the capabilities of the individual components and to show the different variants of a component.

13.4.1 Installing and Configuring Storybook

Storybook comes with a wizard that does most of the setup work for you. To start it, you need to enter the `npm sb init` command in the root directory of your library. The wizard makes sure that all necessary dependencies are installed and creates a new directory named `.storybook` where the configuration for Storybook resides. It also creates a new subdirectory named `stories` in the `src` directory where you can find examples of Storybook stories. You can also delete this directory if you no longer need it for inspiration purposes.

The wizard also adds two scripts to your `package.json` file: `storybook` and `build-storybook`. The `storybook` script starts the Storybook process at port 6006, where you can

view your stories. The second script, `build-storybook`, builds Storybook in a static version that you can deploy to a web server and thus make available to others.

By default, Storybook uses the outdated version 4 of Webpack. If you want to use the newer version 5, you need to run the `npm i -D @storybook/builder-webpack5 webpack` command in the root directory of your library and customize the `main.js` file in the `.storybook` directory by adding webpack 5 as a builder. You can see what this looks like in [Listing 13.21](#):

```
module.exports = {
  core: {
    builder: 'webpack5',
  },
  stories: [
    './src/**/*.stories.mdx',
    './src/**/*.stories.@(js|jsx|ts|tsx)',
  ],
  addons: [
    '@storybook/addon-links',
    '@storybook/addon-essentials',
    '@storybook/addon-interactions',
  ],
  framework: '@storybook/react',
};
```

Listing 13.21 Customization of the Storybook Configuration for Webpack 5
(library/.storybook/main.js)

Whether you run Storybook with Webpack 4 or 5, you can start the process via the `npm run storybook` command from the command line. After that, your system's default browser will open, and you'll see the user interface of Storybook. The next step is to make sure that a story exists for your `Button` component.

13.4.2 Button Story in Storybook

In the default configuration, Storybook automatically includes all files ending with `.stories.tsx` in the user interface. Thus, to create a new story, all you need to do is create a new file and implement the required source code. For the `Button` component, you create a new file named `Button.stories.tsx` in the `src/components/Button` directory. In that file, paste the source code from [Listing 13.22](#):

```
import React from 'react';
import { ComponentStory, ComponentMeta } from '@storybook/react';
import Button from './Button';

export default {
```

```

    title: 'Button',
    component: Button,
  } as ComponentMeta<typeof Button>;

const Template: ComponentStory<typeof Button> = (args) => <
  <Button {...args} />;

export const SimpleButton = Template.bind({});
SimpleButton.args = {
  children: 'click me',
  onClick: () => {},
};

```

Listing 13.22 Storybook Story for the “Button” Component (library/src/components/Button/Button.stories.tsx)

The default export is used to specify how Storybook should display the stories. In addition, you can store other information in this metadata object, such as for Storybook add-ons.

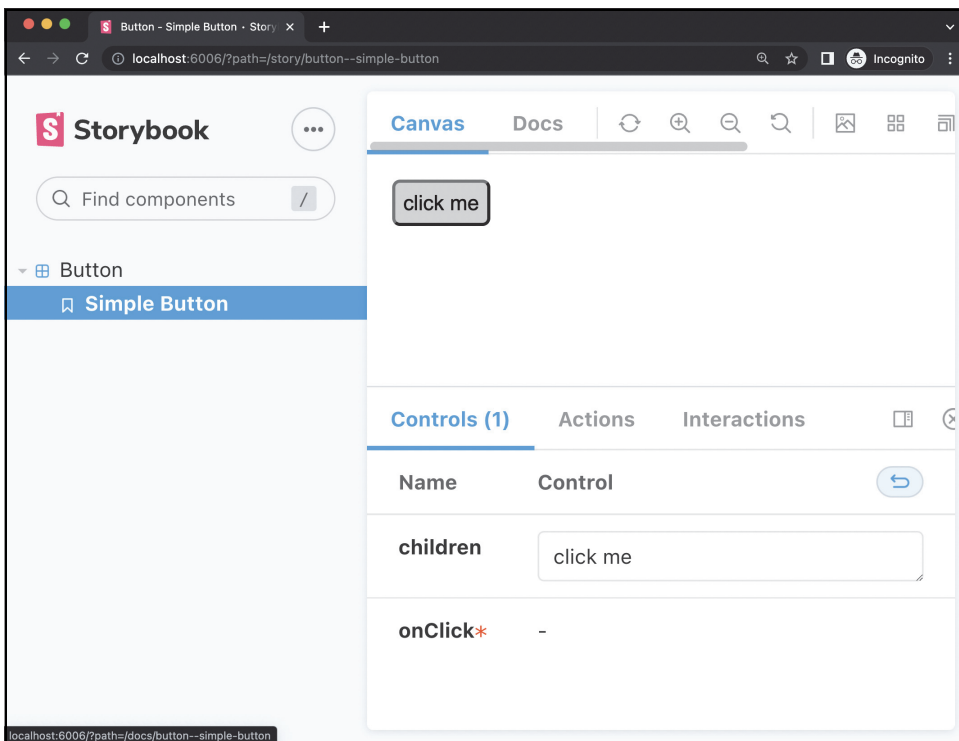


Figure 13.2 Viewing the Button Story in Storybook

The template is the blueprint for the stories in this file. You can customize the template for each story you define—for example, by passing different props. You can see how this works using the `SimpleButton` component as an example. You call the `bind` method of the template to create a new instance, and then define the props. If you run the `npm run storybook` command in the command line and then switch to the browser, you'll see your component in an output like the one shown in [Figure 13.2](#).

13.5 Summary

In this chapter, you learned how to implement your own React library and make it available to other people or applications. Specifically, you learned the following:

- You now know how to initialize a library and what role the different configuration files like *package.json* or *tsconfig.json* play.
- With Rollup, you've been introduced to a lightweight alternative to Webpack as a bundler for your library.
- Although React does not specify file system-level structures, it's recommended to establish a consistent file and directory structure. For example, you can store components and custom hooks in different hierarchies. For each component or hook, there is another subdirectory where you place the associated files.
- You can define and export components and custom hooks just like in an ordinary React application. One of the biggest differences is that you should make sure to export all the structures in the central entry file so that the library is well usable.
- With Rollup and the specific configuration of your library, you can build the library.
- You can include your library in an application using `npm link` or `npm install`. For the installation, you can use different variants like the file system, a web server, or a package registry.
- You should secure the components of your library by using automated unit tests. For this purpose, you use Jest and the React Testing Library.
- Storybook also enables you to showcase your components independent of a specific React application.

The next chapter is dedicated to central state management with the Redux library. There you will learn how to structure and implement even extensive applications.

Chapter 14

Central State Management Using Redux

The term *state* has always been used in connection with the state of a component. This state can contain data that you want the component to display (such as an array of objects that you display in a table) and it can store visual states, such as whether a dialog is open or closed. If multiple components access the same information, you have to make sure that this part of the state is pushed up in the component tree so far that you can pass it on to the corresponding child components via props. An alternative is to make the state available via a context instead of passing it via props.

If you use the combination of state and props, this has several disadvantages. For example, the props are also passed on by uninvolved intermediate components. This can result in many props, only a small number of which involve the component itself. Not only does this affect readability and maintainability, but also reusability. By passing the props around, you also create dependencies that reinforce the coupling of the components, which is something you should really avoid.

Using the Context API is also not without problems: React does not prescribe any form here, and in the worst case you define an unmanageable number of context objects that make maintenance and debugging much more difficult.

The way out of this is *centralized state management*, in which theoretically all components have read access to a state. However, manipulation of this state is only possible via predefined interfaces and a strict workflow so that the connected components can be notified of changes and the view is automatically updated. A very popular architectural pattern in this context is the Flux architecture developed by Facebook.

14.1 The Flux Architecture

The Flux architecture describes the components of a central state management in a component-based application as well as the data flows in such an application. The description is kept abstract enough so that the architectural pattern can be applied to almost any frontend framework, not just to React. Before you add Redux to your application, let's first look at the elements of the Flux architecture.

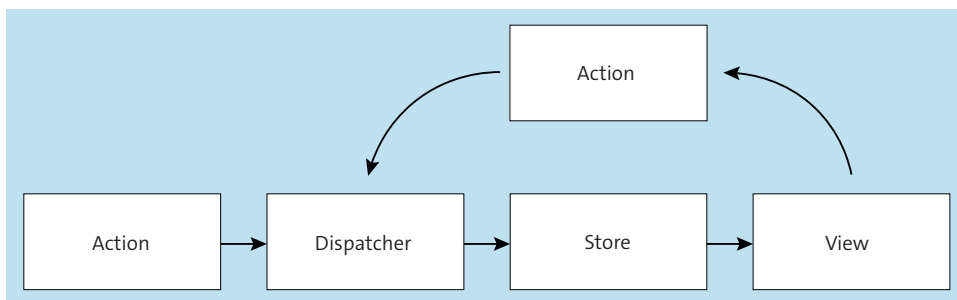


Figure 14.1 Flux Architecture

Figure 14.1 shows a graphical representation of the architecture. Let's start with the central element of the architecture: the store.

14.1.1 The Central Data Store: The Store

Central state management is based on the central storage of data. The structure responsible for this is referred to as the *store*: that's nothing more than an object structure that the application keeps updating at runtime and from which the components read their information.

The Flux architecture provides for multiple stores in one application. This way, you can strictly separate the individual specialties. Some concrete implementations of the architecture combine these decoupled stores into one structure. One example of this is Redux. Fragmentation is recommended to keep the store clear even for larger applications. In this chapter, you'll learn about a strategy for this fragmentation. At this point, this much should be said: the store is divided into different areas, which are managed separately from each other as far as possible. A simple spillover from one section to another isn't readily possible.

The structure of the store consists of objects, arrays, and primitive types. You should avoid storing functions in the store. It's better to swap them out to a separate library so that you only keep data in the store.

14.1.2 Displaying the Data in the Views

As you already know, a React application is made of multiple, self-contained components. These form a tree structure and thus the view layer of the application. A component only takes care of the presentation of a certain aspect of the entire user interface and provides users with an interface for interaction. The components themselves contain only simple view logic. All other functions that are required to manipulate data or that affect the application logic are passed to the components via props. This makes the components independent of the application logic and significantly increases the potential for reusability. Your application thus consists of several layers: a display

layer, which is composed of a component library and aggregation components; and a data storage and logic layer, which is decoupled from the representation.

An important limitation of the Flux architecture is that components have read-only access to the store and cannot modify the data directly. Using this mechanism, the Flux architecture defines the data flow in the application. Changes can only be made in a defined and controlled way. This also ensures that the components are notified of changes and can redisplay themselves.

14.1.3 Actions: The Description of Changes

Actions are simple JavaScript objects that describe the changes to the store. Users can trigger these actions through interactions, for example. One typical action in a Flux application is the creation of a new data record. In this case, the object must contain the information about the type of change—that is, adding the record—and the data of the new record. When developing larger applications, you should make sure that you keep the structure of the actions consistent across the entire application. For this purpose, it's recommended to use the Flux standard action.

The Flux Standard Action

An application processes actions to make the changes to the store. To simplify this processing, actions must have a uniform structure. The *Flux standard action* describes such a structure with the goal of being as readable, simple, and useful as possible for humans. For this purpose, there are several criteria a Flux standard action meets. These are divided into mandatory and optional criteria.

The mandatory criteria are as follows:

- The action must be a simple JavaScript object.
- The object must have a `type` property that specifies what type of action it is. The value of this property is a string that describes the action. In the example of creating a new data record, the value of the `type` property could be `CREATE_BOOK`, for example.

The optional criteria are as follows:

- The action object can have an `error` property. This is of Boolean type. The value `true` means that it is the representation of an error. This is specified in more detail in the `payload` property.
- You can use the `payload` property to submit the action's payload. The value of this property can be both a primitive data type and a composite type such as an array or an object. When the new data record is created, the value of the `payload` property is an object representation of the new data record.
- The third optional property is named `meta` and can contain additional information that further specifies the action but doesn't fit thematically into the `payload` property.

If you stick to this standard, it will be easier to process the actions within the application, because they can be distinguished more easily and you can access the `payload` property directly.

Because actions are simple objects, you can create them directly as object literals. However, so-called action creators are also frequently used. These are factory functions that take care of creating the action objects. You can pass the user data via parameters and receive the action object as a return value that you can use for further work.

14.1.4 The Dispatcher: The Interface between Actions and the Store

The *dispatcher* is the central control element for the data flow in an application. Each store registers a callback function in the dispatcher. When a new action object is dispatched, the appropriate callbacks are executed and the changes are made to the corresponding stores.

As a concrete implementation of the Flux architecture, Redux differs from the original Flux not only in the fact that there is only one large application-wide store instead of multiple stores, but also in the fact that there is no dispatcher. Instead, Redux has so-called reducers.

The Redux Reducer

Because Redux does not have a dispatcher, you need alternative structures to accept action objects and modify the store. This is where the reducer comes into play. You can dispatch an action object using the `dispatch` function, which passes Redux to the reducer function. An application can contain any number of these pure reducer functions. They are called with the current state of the store and the triggered action and return the new version of the state.

At its core, a reducer consists of a more or less extensive `switch-case` statement that checks the type of incoming action and modifies the store accordingly. The return value of the reducer is the new state of the store. If you don't want the reducer to respond to the action, it must still return a value. In this case, that's the unchanged value of the store. The default branch of the `switch-case` statement is usually used for this purpose.

With these elements, you now know the components of the Flux architecture and how they interact, and you've also already experienced some of the special features of Redux. The next step is to include Redux in your application and manage the state of your application using this library.

14.2 Installing Redux

Redux is basically independent of a concrete frontend framework, but it's often used together with React. The source code of the library is maintained in the GitHub repository at <https://github.com/reduxjs/redux> and is available as an NPM package. In addition to Redux, there is the Redux Toolkit, a tool that standardizes the work with Redux and takes work off your hands. For linking Redux and React, an additional package called `react-redux` is available. There are two ways to set up an application: you can either add Redux to an existing React app, or start with the Redux template from Create React App, which already does the basic setup for you and is the recommended structure. You can use the `npx create-react-app library --template redux-typescript` command to initialize your new Redux application.

One of the biggest points of criticism about Redux is that the library adds a lot of overhead to the development of an application. And indeed, Redux needs some structures in order to work properly. For small applications with a small set of features, Redux is usually the wrong choice. But as soon as the application has a certain scope of functionality and you need to access information from multiple places (or as soon as you want to avoid passing the data through multiple component layers), it may make sense to use Redux. The Redux team has recognized this problem and developed the *Redux Toolkit*, a tool that makes it much easier for you to develop an application based on Redux. If you already use either the `redux-typescript` template or the `redux` template from Create React App, the Redux Toolkit is already part of your application.

Once the installation is complete, you can start integrating the individual elements into the application step by step, starting with the store.

The Structure of the Application

The Redux templates for Create React App create a basic structure for your app. Redux includes this at a central point—namely, in the `index.tsx` file. It also creates the `app` and `features` subdirectories in the `src` directory. The `app` directory contains key elements of your application, such as the store configuration and hook functions that you need within your application. In the `features` directory, you then implement the structures of your application. These include, for example, components and the feature-specific slices. You can learn more about the individual structures and their implementation in the following sections.

14.3 Configuring the Central Store

The store is one of the most important elements of a Redux application. It provides the components with the data to display. You can create the store using the `configureStore` function from the `@reduxjs/toolkit` package. As the name *central state management*

suggests, you usually integrate the store into your application at a central location. But when you do this, you create the store in a separate file rather than directly in the *index.tsx* file. The Create React App template has created the *store.ts* file in the *src/app* directory for the store. The source code of this file is shown in [Listing 14.1](#):

```
import { configureStore, ThunkAction, Action } from '@reduxjs/toolkit';
import counterReducer from '../features/counter/counterSlice';

export const store = configureStore({
  reducer: {
    counter: counterReducer,
  },
});

export type AppDispatch = typeof store.dispatch;
export type RootState = ReturnType<typeof store.getState>;
export type AppThunk<ReturnType = void> = ThunkAction<
  ReturnType,
  RootState,
  unknown,
  Action<string>
>;
```

Listing 14.1 Creation of the Redux Store (*src/app/store.ts*)

In the example, you can see that you can pass an object with a *reducer* property to the *configureStore* function. This function has several properties that represent the individual features of your application. In this case, there is already a *counter* implementation, which we'll replace with a list of books throughout this chapter.

In addition to creating the store, you'll see multiple type definitions in the *store.ts* file that you can use when implementing your application. These are the types of the dispatch function, *RootStates* and *AppThunk*. The last type refers to *Redux Thunk*, an asynchronous middleware for Redux, which we'll describe in more detail in the next chapter.

The store has already been integrated into the application in the *index.tsx* file. React-Redux uses the Context API of React to make the store and other structures available across the entire application. The *react-redux* package provides a context provider that you can integrate into your application. You need to assign the store to the *store* prop of the provider. You can see the source code of the *index.tsx* file in [Listing 14.2](#):

```
import React from 'react';
import { createRoot } from 'react-dom/client';
import { Provider } from 'react-redux';
import { store } from './app/store';
```

```
import App from './App';
import reportWebVitals from './reportWebVitals';
import './index.css';

const container = document.getElementById('root');
const root = createRoot(container);

root.render(
  <React.StrictMode>
    <Provider store={store}>
      <App />
    </Provider>
  </React.StrictMode>
);

reportWebVitals();
```

Listing 14.2 Integration of the Store into the Application (src/index.tsx)

In the store, you can register other elements besides the reducers, which are referred to as *middleware*. These are extensions that allow you to implement functions which are executed between the dispatching of an action and the handling of that action by the reducer. One very popular middleware option for Redux is *Redux Dev Tools*, which allows you to debug a Redux application.

Redux Dev Tools consists of two components: middleware that you integrate directly into your application, and the actual browser extension that you need to install in your browser.

14.3.1 Debugging Using the Redux Dev Tools

Redux Dev Tools is a browser extension that allows you to look at your Redux application at runtime. You can access the store and view the values. In addition, you can see over time which actions were dispatched, which values they contain, and how they affected the state of the application.

Redux Toolkit already contains this extension for Redux, and you don't need to do anything to access it other than insert the `devTools` property with a value of `true` into the object you pass to the `configureStore` function. [Listing 14.3](#) shows the source code of the `store.ts` file:

```
...
export const store = configureStore({
  reducer: {
    counter: counterReducer,
  },
});
```

```

    devTools: true,
  });
  ...

```

Listing 14.3 Configuration of the Redux Store for the Development Operation (src/app/store.ts)

Once you've finished configuring the store, you still need to install the extension in your browser. You can do this by searching for Redux Dev Tools in your browser's extension store and installing them. After a successful installation, you'll have the **Redux** option available in the developer tools of your browser. [Figure 14.2](#) shows a screenshot of Redux Dev Tools for the current state of the application.

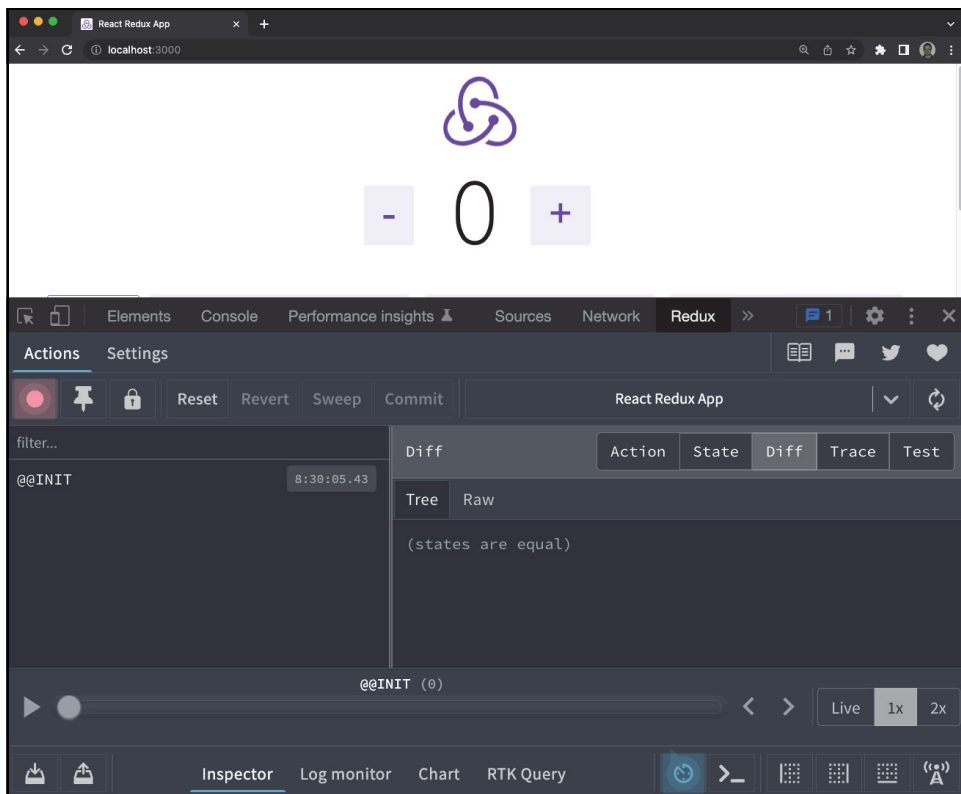


Figure 14.2 Redux Dev Tools

In the left-hand section of Redux Dev Tools, you can see a list of dispatched actions. On the right you can see more information about the state of the application at the time when the action was dispatched.

As you can see here, the list of actions is empty except for the initial entry. This must be changed in the next steps. The first step consists of the initial filling of the store. You can achieve this by implementing reducers for your application.

14.4 Handling Changes to the Store Using Reducers

You can use the sample application we implement in this chapter to manage books—so one feature of the application is named `Books`. Because Redux allows you to fragment the store, the feature gets its own *slice*, which is a separate section of the store. It also has its own reducer and actions. As you have already seen, you integrate all slices in the *store.ts* file in a central location.

14.4.1 The Books Slice

First, you create another directory named *books* in the *src/feature* directory and place the *booksSlice.ts* file there. This file contains the initial state of the substate, the reducers, and the actions.

Side Effects in Redux

Redux focuses on central state management within your application—that is, the management of the state. This should basically be free of side effects. A typical side effect is the communication with a server. The treatment of such side effects is not part of Redux. For this purpose, you use middleware components. You can either implement these yourself or use existing solutions, a selection of which you'll learn about in the next chapter. In the remainder of the chapter, you'll implement a local state management version for your application, which you reconnect to the server in the next chapter.

Before you start implementing the structures of your new slice, you first define the book type in the *book.ts* file in the *src/feature/books* directory:

```
export type Book = {
  id: number;
  title: string;
  author: string;
  isbn: string;
  rating?: number;
};

export type InputBook = Omit<Book, 'id'> & {
  id?: number;
};
```

Listing 14.4 Definition of the “Book” Type (*src/features/books/book.ts*)

Because we won't deal with server communication until the next chapter, you need an initial state of the store so that you can display something without having first to create data records on each reload. This task is performed by the *booksData.ts* file in the *src/featre/books* directory. The file exports an array of objects that would normally be provided by the server. You can see the contents of the file in [Listing 14.5](#):

```
import { Book } from './Book';

const booksData: Book[] = [
  {
    id: 1,
    title: 'JavaScript - The Comprehensive Guide',
    author: 'Philip Ackermann',
    isbn: '978-3836286299',
  },
  {
    id: 2,
    title: 'Clean Code',
    author: 'Robert Martin',
    isbn: '978-0132350884',
  },
  {
    id: 3,
    title: 'Design Patterns',
    author: 'Erich Gamma',
    isbn: '978-0201633610',
  },
];

export default booksData;
```

Listing 14.5 Initial State for the Books Slice (*src/features/books/booksData.ts*)

The next step is to implement the slice in the *booksSlice.ts* file. The implementation is shown in [Listing 14.6](#):

```
import { createSlice } from '@reduxjs/toolkit';
import { Book } from './Book';
import booksData from './booksData';

export type BooksState = Book[];

export const booksSlice = createSlice({
  name: 'books',
  initialState: { books: booksData },
  reducers: {},
});

export default booksSlice.reducer;
```

Listing 14.6 Basic Implementation of “booksSlice” (*src/features/books/booksSlice.ts*)

You use the `createSlice` function from the `@reduxjs/toolkit` package to create the new slice. In the minimal implementation, you pass the function an object with the name of the slice in the `name` property, the initial state via the `initialState` property, and an empty `reducers` object. For easier integration, you export the `reducer` property of the slice via a default export.

14.4.2 Integration of “BooksSlice”

A Redux application consists of at least one slice but can have any number of these structures. For Redux to know about the existence of each slice, you must integrate the slice's reducer in the store. You can do this in the `store.ts` file in the `src/app` directory. In [Listing 14.7](#), you can see the adjusted source code of this file:

```
import { configureStore, ThunkAction, Action } from '@reduxjs/toolkit';
import counterReducer from '../features/counter/counterSlice';
import booksReducer from '../features/books/booksSlice';

export const store = configureStore({
  reducer: {
    counter: counterReducer,
    books: booksReducer,
  },
  devTools: true,
});

export type AppDispatch = typeof store.dispatch;
export type RootState = ReturnType<typeof store.getState>;
export type AppThunk<ReturnType = void> = ThunkAction<
  ReturnType,
  RootState,
  unknown,
  Action<string>
>;
```

Listing 14.7 Integration of “BooksReducer” (`src/app/store.ts`)

To include the slice, in the object you pass to the `configureStore` function, you add a new `books` property in the `reducer` object that references the reducer of `BooksSlice`.

Once you’ve implemented this adjustment, you can open Redux Dev Tools in development mode and inspect the current state of your application. The output should look as shown in [Figure 14.3](#).

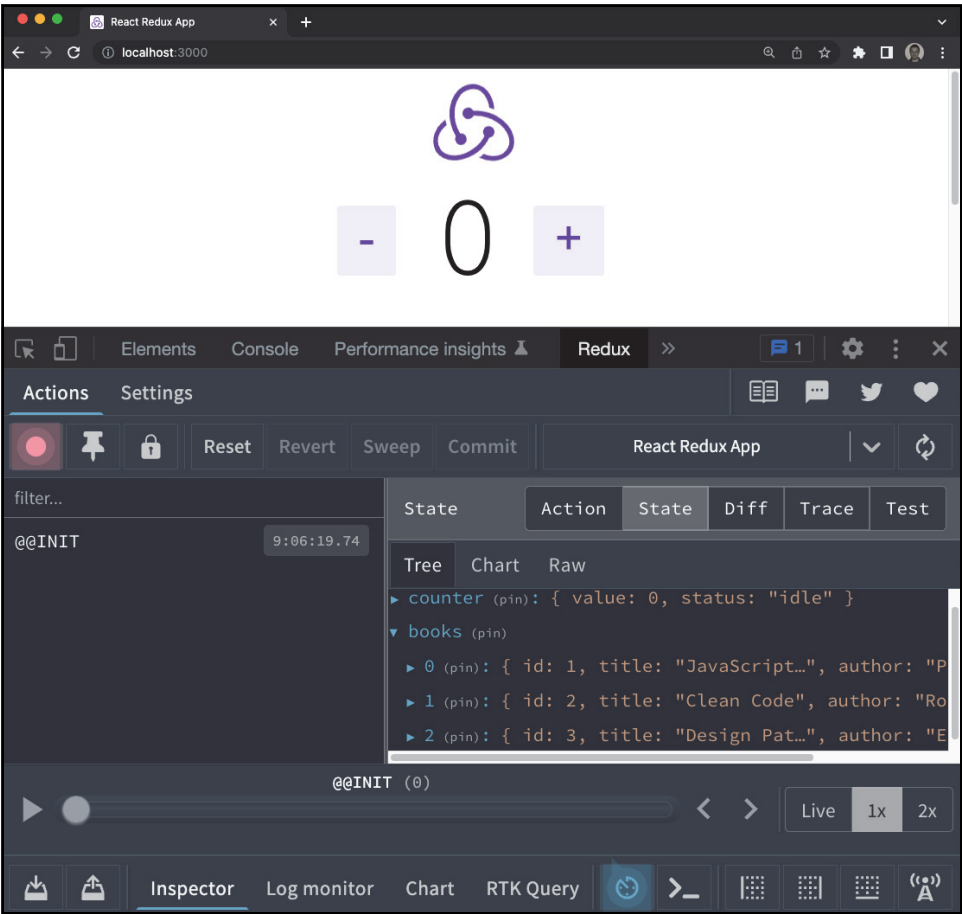


Figure 14.3 Initial State of the Application after the Reducer Integration

When you select the **State** tab in Redux Dev Tools on the right-hand side, you'll see two slices: `counter` and `books`. If you activate the `books` slice, you'll see the three records that you defined as the initial value. This means that Redux is correctly integrated into your application. The browser still displays the default app that you created with Create React App. But we'll take care of that in the next step.

14.5 Linking Components and the Store

The architecture behind Redux allows the React components of an application to be able to access the store on a read-only basis. To do this, you use the provider from the `react-redux` package to make the store available within the entire application via the Context API of React.

14.5.1 Displaying the Data from the Store

To display the data, you implement a new component named `List` in the `List.tsx` file in the `src/features/books` directory. [Listing 14.8](#) contains the implementation of the component:

```
import React from 'react';
import { useSelector } from 'react-redux';
import { RootState } from '../app/store';

const List: React.FC = () => {
  const books = useSelector((state: RootState) => state.books);

  return (
    <table>
      <thead>
        <tr>
          <td>Title</td>
          <td>Author</td>
          <td>ISBN</td>
        </tr>
      </thead>
      <tbody>
        {books.map((book) => (
          <tr key={book.id}>
            <td>{book.title}</td>
            <td>{book.author}</td>
            <td>{book.isbn}</td>
          </tr>
        ))}
      </tbody>
    </table>
  );
};

export default List;
```

Listing 14.8 Display of Data from the Store in the “List” Component
(`src/features/books/List.tsx`)

The `useSelector` hook from the `react-redux` package enables you to access the store in read-only mode from within the component. Thanks to typing the structures, you don't need to pay attention to anything else in the component and can directly use the data the selector hook returns to you.

The JSX structure returned by the component is a simple table that displays the titles, authors, and ISBNs of the data records. To enable you to view the information in the browser, you integrate the `List` component into the `App` component of your application, as shown in [Listing 14.9](#):

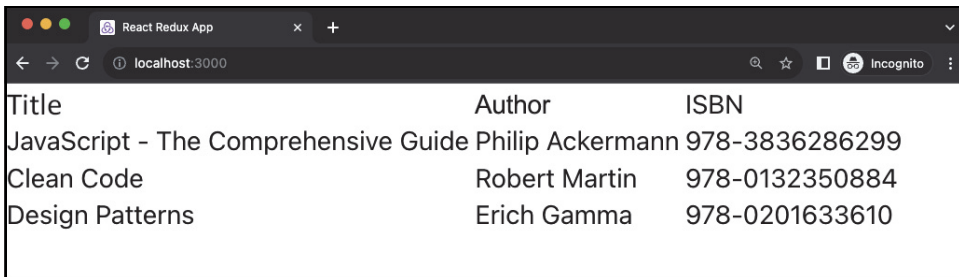
```
import React from 'react';
import './App.css';
import List from './features/books/List';

const App: React.FC = () => {
  return <List />;
};

export default App;
```

Listing 14.9 Integration of the “List” Component into the “App” Component (src/App.tsx)

Once your development process has been started via the `npm start` command, you can switch to the browser. There you’ll see the table with the individual data records, as shown in [Figure 14.4](#).



The screenshot shows a web browser window titled "React Redux App" with the address bar displaying "localhost:3000". The browser is in Incognito mode. The main content area displays a table with three columns: Title, Author, and ISBN. The table contains three rows of data:

Title	Author	ISBN
JavaScript - The Comprehensive Guide	Philip Ackermann	978-3836286299
Clean Code	Robert Martin	978-0132350884
Design Patterns	Erich Gamma	978-0201633610

Figure 14.4 Display of the Data from the Store

14.5.2 Selectors

As you saw in the previous example, you do not access a state object directly, but use selectors for it. In the simplest case, such a *selector* is a function that receives the store object and returns a specific piece of information from it. You can define these selectors directly in the component, as in the case of the `List` component. However, a more elegant solution is to define these selectors in the slice. The advantage of this approach is that you can reuse the selectors and thus have fewer duplicates in the code. In the selector function, you can access the entire state of the application, which is represented by the state object. The section where the books records are stored is named `books`, and the data itself resides in the `books` property. So the full path is `state.books.books`.

```
import { createSlice } from '@reduxjs/toolkit';
import { RootState } from '../app/store';
import { Book } from './Book';
import booksData from './booksData';

export type BooksState = {...};

export const booksSlice = createSlice({...});

export const selectBooks = (state: RootState) => state.books.books;

export default booksSlice.reducer;
```

Listing 14.10 Definition of the “selectBooks” Selector (src/features/books/booksSlice.ts)

You can then use the selector defined in [Listing 14.10](#) in your component, as shown in [Listing 14.11](#):

```
import React from 'react';
import { useSelector } from 'react-redux';
import { selectBooks } from './booksSlice';

const List: React.FC = () => {
  const books = useSelector(selectBooks);

  return (<table>... </table>);
};

export default List;
```

Listing 14.11 Using the “selectBooks” Selector (src/features/books/List.tsx)

However, selectors can do considerably more than just encapsulate the paths to the desired information.

Reducing Redundancies in the Store Using Selectors

When dealing with the store of your application, you should make sure that a certain piece of information is available only once. The reason for this is that otherwise you will have to worry about updating multiple places, which is a potential source of error for inconsistencies in the store.

Rule

If information can be calculated from the store data, you should do this in the selector instead of storing the information redundantly in the store.

Typically, filters or sorting, for example, provide for the situations described. In this case, you would keep a filtered list and the original version of the list. Another example of avoidable redundancy of information in the store is when you want to display the number of records in the books list. This information is already available as the `length` property of the `books` array. At this point, it's better to use a selector that makes it easier for you to access this information.

Higher-Order Selectors

For static selectors that always return the same value, the implementation is clear. Filters or similar dynamic operations can be implemented much more elegantly and without unnecessary redundancies. For this purpose, you use higher-order functions as selectors. These are functions that return the actual selector function and that you can parameterize. [Listing 14.12](#) shows how to extend the `booksSlice.ts` file with the `selectBook` selector. With this, you can read a single book from the store and use it for the form, for example:

```
import { createSlice } from '@reduxjs/toolkit';
import { RootState } from '../app/store';
import { Book, InputBook } from '../Book';
import booksData from './booksData';

export type BooksState = {...};

export const booksSlice = createSlice({...});

export const selectBooks = (state: RootState) => state.books.books;

export function selectBook(state: RootState): (id?: number) => InputBook {
  return (id?: number): InputBook => {
    const book = selectBooks(state).find((book) => book.id === id);
    if (!book) {
      return {
        title: '',
        author: '',
        isbn: '',
      };
    }
    return book;
  };
}

export default booksSlice.reducer;
```

Listing 14.12 Higher-Order Selector for Selecting a Single Data Record
(src/features/books/booksSlice.ts)

The `selectBook` selector gets a reference to `RootState` and returns a function itself. This inner function accepts the ID of the data record being searched for and uses this information to search for the matching record in the state. To access the records, you use the `selectBooks` selector. This has the advantage that you only have to make adjustments in one place if the path in the store should change.

Selectors make it easier to access parts of the store and greatly simplify refactorings of the state structure. Because selectors have been established as a design tool for several years now, a large number of libraries have been created to make your work easier. One of the most popular solutions in this area is *Reselect*.

14.5.3 Implementing Selectors Using Reselect

The Reselect library is maintained in the same GitHub repository as Redux and is an addition to Redux. The `createSelector` function of Reselect is an integral part of the Redux Toolkit, so you don't need to install anything else. Reselect is based on simple selector functions, like those you've already implemented with the `selectBooks` function. Reselect extends these functions in such a way that the functions are *memoized*, which means that the return values are cached until the arguments change.

It makes sense to integrate this library especially if you use computationally intensive calculations to determine the state. Typically, these are filters or sort functions. The selectors you create using Reselect only show their strength when you use them more than once. For example, if you implement a filter to find books with a certain rating, you can implement a selector that will give you the data records.

If you implement a dynamic selector, it needs the information about what state the application currently is in. This tells us what the expected output value of the selector is. In this example, you store the information about which rating you're interested in in the `ratingFilter` property in the `BooksState` of the `BookSlice`. You can specify any number from 0 to 5 as the initial value here. The value 0 stands for a deactivated filter, and the numbers 1 to 5 ensure that you only receive books with the respective rating.

On this basis, you can now implement the selector in the next step. This is based on the values of books as well as the `ratingFilter` property of `BooksState`. For this reason, you first create another selector for the `ratingFilter` property. Then you use the `createSelector` function of Reselect to create the actual selector, as shown in [Listing 14.13](#):

```
import { createSelector, createSlice } from '@reduxjs/toolkit';
import { RootState } from '../app/store';
import { Book, InputBook } from './Book';
import booksData from './booksData';

export type BooksState = {...};

export const booksSlice = createSlice({...});
```

```
export const selectBooks = (state: RootState) => state.books.books;
export const selectRatingFilter = (state: RootState) =>
  state.books.ratingFilter;

export const selectByRating = createSelector(
  [selectBooks, selectRatingFilter],
  (books, ratingFilter) => {
    if (ratingFilter === 0) {
      return books;
    }
    return books.filter((book) => book.rating === ratingFilter);
  }
);

export function selectBook(state: RootState): (id?: number) => InputBook {...}

export default booksSlice.reducer;
```

Listing 14.13 Selector for Filtering the Books by Rating (src/features/books/booksSlice.ts)

The `createSelector` function receives an array of selector functions as its first argument. Based on these values, the caching of the return values takes place. This means that the function generates the same output for the same input and obtains it from its internal cache. So in the example, the `filter` function is executed only the first time. If you ask for the same values again later, they will be delivered faster.

The more complex the functions for calculating the values, the more worthwhile it is to use `Reselect`. However, this also means that `Reselect` provides no real benefit for selectors that are called with different values each time they are called and therefore always return different values.

Until now, you've only had read access to the store, as writing to the store isn't possible in a direct way. On the following pages, you'll learn how to manipulate the store by using actions. I'll demonstrate this using the CRUD operations of the books list as an example.

14.6 Describing Changes with Actions

Actions are simple JavaScript objects that are sent to the reducer using the `dispatch` function. Based on the structure of the action, the reducer decides in which way the store of the application should be manipulated. Usually, a `type` property in the action object defines the type of the action. The simplest operation to implement when managing data records in an application is deleting records. The action for deleting receives only the ID of the data record to be deleted as user data.

When integrating actions into your application, you proceed through several steps:

1. You write a reducer function in the `reducer` property of your slice.
2. You export the respective action creator function from the `actions` property of the slice.
3. You use the `dispatch` function, which you get from the `useAppDispatch` function, to trigger the action you created via the action creator function.
4. Redux processes the action via the appropriate reducer function and updates the store.

Most of the adjustments for the modification of the store involve the slice file. [Listing 14.14](#) shows the adjustments to the `booksSlice.ts` file:

```
import { createSelector, createSlice, PayloadAction } from '@reduxjs/toolkit';
import { RootState } from '../app/store';
import { Book, InputBook } from './Book';
import booksData from './booksData';

export type BooksState = {...};

export const booksSlice = createSlice({
  name: 'books',
  initialState: { books: booksData, ratingFilter: 0 },
  reducers: {
    remove(state, action: PayloadAction<number>) {
      const index = state.books.findIndex(
        (book) => book.id === action.payload
      );
      state.books.splice(index, 1);
    },
  },
});

export const { remove } = booksSlice.actions;

export const selectBooks = (state: RootState) => state.books.books;
export const selectRatingFilter = (state: RootState) =>
  state.books.ratingFilter;

export const selectByRating = createSelector(...);

export function selectBook(state: RootState): (id?: number) => InputBook {...}

export default booksSlice.reducer;
```

Listing 14.14 Implementation of the “remove” Reducer (`src/features/books/booksSlice.ts`)

In the object that you pass to the `createSlice` function, you define a new method named `remove` in the object that is behind the `reducers` property. Redux Toolkit automatically ensures that you're provided with a corresponding action creator function. In the `remove` method, you can access the `state` object that reflects the current state of the slice data. In addition, you can define an action. As type, you use the generic `PayloadAction` type, to which you pass the data type you expect in the `payload` property of the action. When deleting data records, this is the ID of a record—that is, a number.

Inside the reducer function, you find the index of the record you want to delete and remove it using the `splice` method. Normally, you shouldn't manipulate state objects directly under any circumstances; instead, it's better to work on a copy of the object and return it. In this case, however, that isn't necessary because Redux Toolkit uses the Immer library and the state object is thus immutable.

Using the `actions` property of the slice, you can access the action creator function, which has the same name as the `reducers` property—that is, `remove`. You export this action creator function so that you can use it in your components.

With this structure in place, you can now move onto integrating the action into the `List` component:

```
import React from 'react';
import { useSelector } from 'react-redux';
import { useAppDispatch } from '../app/hooks';
import { selectBooks, remove } from './booksSlice';

const List: React.FC = () => {
  const books = useSelector(selectBooks);
  const dispatch = useAppDispatch();

  return (
    <table>
      <thead>...</thead>
      <tbody>
        {books.map((book) => (
          <tr key={book.id}>
            <td>{book.title}</td>
            <td>{book.author}</td>
            <td>{book.isbn}</td>
            <td>
              <button onClick={() => dispatch(remove(book.id))}>
                Delete
              </button>
            </td>
          </tr>
        ))}
      </tbody>
    </table>
```

```
);
};

export default List;
```

Listing 14.15 Integration of the “remove” Action into the “List” Component (src/features/books/List.tsx)

You use the `useAppDispatch` function to create the actual dispatch function, which enables you to trigger the actions. To allow users to interact with your application, you add a button element to each row of the table. In the click handler of this button, you use the action creator function to create an action object that you trigger via the dispatch function. In this state of your application, you can switch to the browser and delete your data records. Although the data records do disappear, the action isn’t persistent because there is no backend to take care of saving yet. If you reload the browser window, the data records will appear again. We’ll deal with the topic of persistence in the next chapter. For example, if you delete the record with ID 2, the action creator function creates an object as shown in [Listing 14.16](#):

```
{
  "type": "books/remove",
  "payload": 2
}
```

Listing 14.16 Structure of an Action Object

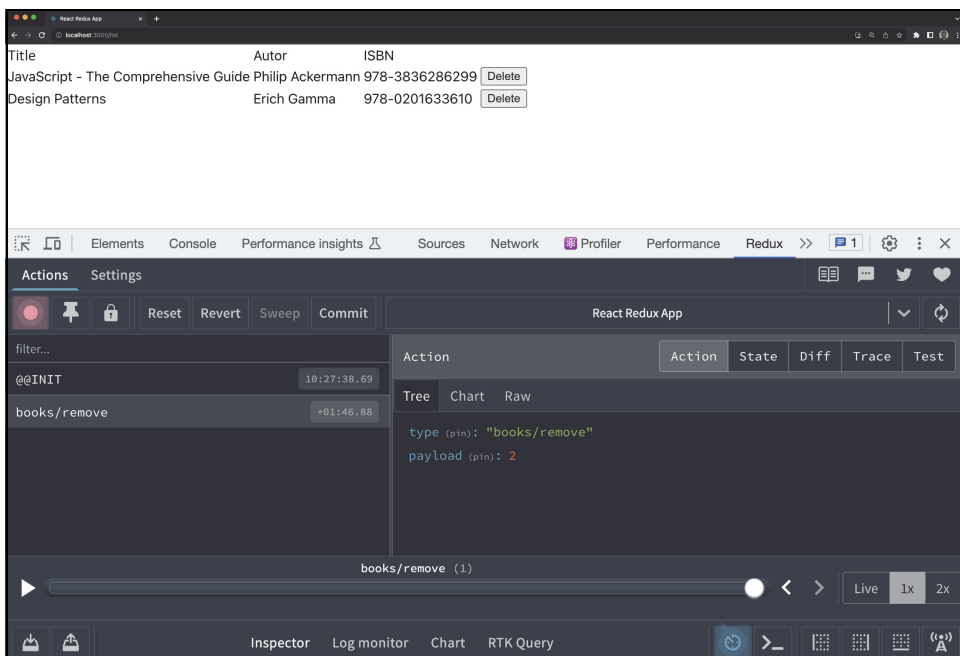


Figure 14.5 Action in Redux Dev Tools

The type of the action object is composed of the names of the slice and the reducer. The value of the payload property comes from the value you used to call the action creator function. When you switch to the browser and open Redux Dev Tools, you can watch the action being recorded when you press one of the buttons. [Figure 14.5](#) shows an example of this view.

Aside from deleting the data, you still need to take care of creating and editing records.

14.7 Creating and Editing Data Records

The first step for the two remaining write operations is to add a save method to the slices in the reducers. In this method, you take care of both creating and modifying existing data records. [Listing 14.17](#) contains the source code:

```
import { createSelector, createSlice, PayloadAction } from
  '@reduxjs/toolkit'; ←
import { RootState } from '../app/store';
import { Book, InputBook } from './Book';
import booksData from './booksData';

export type BooksState = {...};

export const booksSlice = createSlice({
  name: 'books',
  initialState: { books: booksData, ratingFilter: 0 },
  reducers: {
    remove(state, action: PayloadAction<number>) {...},
    save(state, action: PayloadAction<InputBook>) {
      if (action.payload.id) {
        const index = state.books.findIndex(
          (book) => book.id === action.payload.id
        );
        state.books[index] = action.payload as Book;
      } else {
        const nextId = Math.max(...state.books.map( ←
          (book) => book.id)) + 1;
        state.books.push({ ...action.payload, id: nextId });
      }
    },
  },
});

export const { remove, save } = booksSlice.actions;
```

```

export const selectBooks = (state: RootState) => state.books.books;
export const selectRatingFilter = (state: RootState) =>
  state.books.ratingFilter;

export const selectByRating = createSelector(...);

export function selectBook(state: RootState): (id?: number) => InputBook {...}

export default booksSlice.reducer;

```

Listing 14.17 Implementation of the “save” Reducer (src/features/books/booksSlice.ts)

Like the `remove` method, the `save` method of the `reducers` method receives the current state and an action object. However, this action object is of type `PayloadAction<InputBook>`, which means you expect an object of type `InputBook` as payload. Within the method, you distinguish whether the data record is new or existing based on the presence of the `id` property.

For an existing record, you need to find the index of the record and overwrite the value with the changed record. For a new data record, you create a new ID value by finding the highest value that was assigned previously and adding the value 1. Then you push the data record into the state along with the new ID. Again, Immer ensures that you can modify the structures directly and don't need to worry about cloning them.

The integration into the application is a bit more complex because you have to implement a form and integrate that into the application. To make your work a little easier and at the same time improve the usability of the application, you install React Router and React Hook Form via the `npm install react-router-dom react-hook-form` command. This way, you first define the required routes in your `App` component, assuming that your app already has a `Form` component.

```

import React from 'react';
import './App.css';
import List from './features/books/List';
import Form from './features/books/Form';
import { BrowserRouter, Routes, Route, Navigate } from 'react-router-dom';

const App: React.FC = () => {
  return (
    <BrowserRouter>
      <Routes>
        <Route path="/edit/:id" element={<Form />} />
        <Route path="/new" element={<Form />} />
        <Route path="/list" element={<List />} />
        <Route path="/" element={<Navigate to="/list" />} />
      </Routes>
    </BrowserRouter>
  );
};

```

```
    </BrowserRouter>
  );
};
```

```
export default App;
```

Listing 14.18 Routing Configuration for the Application (src/App.tsx)

In the App component, you define a total of four routes. The first two routes are used to edit and create data records and to reference the Form component. The /list route is responsible for displaying the List component, and the fourth route takes care of redirecting the requests to the list by using the / path.

In the next step, you implement the Form component in the *src/features/books* directory. [Listing 14.19](#) contains the corresponding source code:

```
import React, { useEffect } from 'react';
import { useSelector } from 'react-redux';
import { save, selectBook } from './booksSlice';
import { useForm } from 'react-hook-form';
import { useNavigate, useParams } from 'react-router-dom';
import { useAppDispatch } from '../../app/hooks';
import { InputBook } from './Book';

const Form: React.FC = () => {
  const getBook = useSelector(selectBook);
  const dispatch = useAppDispatch();
  const navigate = useNavigate();
  const { register, handleSubmit, reset } = useForm<InputBook>({
    defaultValues: { title: '', author: '', isbn: '' },
  });

  const { id } = useParams<{ id?: string }>();

  useEffect(() => {
    if (id) {
      const book = getBook(parseInt(id, 10));
      reset(book);
    }
  }, [id, reset, getBook]);

  return (
    <form
      onSubmit={handleSubmit((data) => {
        dispatch(save(data));
      })}
```

```

        navigate('/list');
      })}
    >
    <div>
      <label htmlFor="title">Title:</label>
      <input type="text" {...register('title')} />
    </div>
    <div>
      <label htmlFor="author">Author:</label>
      <input type="text" {...register('author')} />
    </div>
    <div>
      <label htmlFor="isbn">ISBN:</label>
      <input type="text" {...register('isbn')} />
    </div>
    <div>
      <button type="submit">save</button>
    </div>
  </form>
);
};

```

```
export default Form;
```

Listing 14.19 Implementation of the “Form” Component (src/features/books/Form.tsx)

In the `Form` component, you use the `selectBook` selector to create the `getBook` function, which enables you to search for a single data record. You also create the dispatch function with the `useAppDispatch` hook and the `navigate` function with the `useNavigate` hook from the `react-router-dom` package. In the next step, you access the `id` property in the URL path. This is optional because you don’t pass an ID with the `/new` route. Before defining the JSX structure, you fetch the appropriate book record and set its values into the form using the `reset` function from `React Hook Form`.

Using `React Hook Form`, you then define a form that contains three input fields: **Title**, **Author**, and **ISBN**. We do not perform a validation in this example. You also add a **Submit** button to save the record.

In the submit handler of the form, you combine `Redux` with `React Hook Form` and `React Router` by dispatching the `save` action with the changed or new data record, respectively, and redirecting it to the `/list` path via `React Router`.

To test your form in the browser, you still need to add references to the `/edit/:id` and `/new` paths in the `List` component. The adjusted version of the `List` component is shown in [Listing 14.20](#):

```
import React from 'react';
import { useSelector } from 'react-redux';
import { useNavigate } from 'react-router-dom';
import { useAppDispatch } from '../../app/hooks';
import { selectBooks, remove } from './booksSlice';

const List: React.FC = () => {
  const books = useSelector(selectBooks);
  const dispatch = useAppDispatch();
  const navigate = useNavigate();

  return (
    <>
      <table>
        <thead>
          <tr>
            <td>Title</td>
            <td>Author</td>
            <td>ISBN</td>
            <td></td>
            <td></td>
          </tr>
        </thead>
        <tbody>
          {books.map((book) => (
            <tr key={book.id}>
              <td>{book.title}</td>
              <td>{book.author}</td>
              <td>{book.isbn}</td>
              <td>
                <button onClick={() => dispatch(remove(book.id))}>
                  delete
                </button>
              </td>
              <td>
                <button onClick={() => navigate(`/edit/${book.id}`)}>
                  edit
                </button>
              </td>
            </tr>
          ))}
        </tbody>
      </table>
      <button onClick={() => navigate('/new')}>new</button>
    </>
  );
}
```

```

    </>
  );
};

export default List;

```

Listing 14.20 Integration of the Navigation for the “Form” Component into the “List” Component

14.8 Summary

In this chapter, you learned about Redux, which is one of the most popular extensions for React. If the scope of an application grows beyond a certain point and you access centralized information from multiple locations, it may make sense to implement centralized state management for the application:

- The *Flux architecture* consists of stores, views, actions, and dispatchers, as well as the directed data flow between these elements. The Flux architecture is specific to neither Redux nor React. It is implemented by numerous other libraries for central state management as well.
- *Redux* is a concrete implementation of the Flux architecture and adapts it in some places. For example, there is only one store instead of multiple stores.
- One important tool to keep track of a Redux application is Redux Dev Tools. This is implemented on the one hand as middleware in Redux and on the other hand as a browser extension. This enables you to monitor your application over the runtime and read information about the respective states and events.
- *Redux Toolkit* addresses one of the major weaknesses of Redux as it reduces the overhead created by the structures of the library. This allows you to write separate reducer functions without having to worry about the structure of the action objects.
- You can divide the Redux store of your application into several *slices*. These are independent sections of the store that represent different functional areas of your application. You create such a slice via the `createSlice` function of Redux Toolkit.
- *Selectors* are used to access the parts of the store. These are simple functions that get the state and return a certain piece of information. The selectors can be memoized using libraries such as *Reselect* to improve application performance.
- Write operations do not take place directly in the store, but are encapsulated via actions. For the structure, the Flux standard action form has become established. The `createSlice` function creates an object that has, among other things, the `actions` property. This property provides you with other methods, the so-called action creator functions, which allow you to create actions.

- The *reducer methods* you create in your slices are responsible for modifying the store. You'll get the current state of the store and a reference to the triggered action. Based on this combination, the function generates a new version of the store. Redux ensures that the components affected by the change are re-rendered.

Redux itself is basically designed to work synchronously and thus doesn't cover tasks such as read or write communication with the server. The library doesn't specify how to handle asynchronous operations, which are usually handled by so-called *middleware*. In the next chapter, you'll learn what the term *middleware* means and how you can integrate middleware libraries into your application.

Chapter 15

Handling Asynchronicity and Side Effects in Redux

In [Chapter 14](#), you learned about Redux, one of the most popular libraries for centralized state management in React applications. Redux is an implementation of the Flux architecture that provides structures for read and write access to a central data store. By default, Redux works with synchronous operations and only locally in the browser, but one of the most important aspects of a web frontend is that it communicates with a server to load and store data. Redux doesn't offer a standard solution for this case; instead, it only provides interfaces that enable you to solve this task.

These interfaces are referred to as *middleware* and allow a Redux application to be extended without going too deep into the actual application. But Redux middleware can do more than just handle server communication. In this chapter, you'll first learn what exactly middleware is in Redux and how it's structured. We'll then present various libraries for server communication.

15.1 Middleware in Redux

At its core, Redux middleware is a function that's registered when the store is created. The middleware has access to the store and to the action that was dispatched at the time of the call. However, the middleware can only have read access to the store. Write access is reserved for the reducer. If a change is supposed to be caused by middleware, then this is done by means of dispatching a new action, which is then received by the reducer.

But let's return to the middleware: because it's intended as a third-party interface, numerous implementations already exist in the form of packages that you can install in your application and then integrate. Examples include the following:

- **Redux Dev Tools**

You already learned about this debugging tool in the previous chapter. It takes advantage of the fact that the package can access all the required information as middleware and pass it to the browser extension.

- **Redux Thunk**

This library is just one example out of numerous implementations that solve the problem of asynchronous operations in a Redux application.

You can not only use existing implementations, but also write your own middleware. But in this case, you should avoid reinventing the wheel. Especially in the area of solutions for server communication, there are already a number of mature libraries that you can use. If you need a special solution for your application that's not or only insufficiently covered by the existing packages, there is of course nothing to stop you from developing your own.

The signature of middleware follows a clearly defined structure and consists of three nested functions. [Listing 15.1](#) shows the basic structure of middleware:

```
const customMiddleware = store => next => action => {  
  return next(action);  
}
```

Listing 15.1 Structure of Middleware

The three-part structure of the middleware reflects the individual steps leading to the final middleware: the `customMiddleware` function is called via the `applyMiddleware` function of Redux. The middle function encapsulates the dispatch functionality, and the inner function executes the actual action and dispatches the action further. The three arguments available to you in the middleware have the following meaning:

- **store**

The `store` object represents a reference to the Redux store. You can use this object for read access to the store.

- **next**

This function sends the action to the reducer or to other middleware, which ensures the continuity of the execution chain.

- **action**

The `action` object represents the currently dispatched action and contains all information about the upcoming change.

What's particularly relevant in the implementation besides the `next` function that you get as an argument is the `getState` method of the `Store` object. The combination of the two decides which version of the state of your application you get: If you call the `next` function before `getState`, you access the version of the state after it has been updated by the reducer. If `getState` is executed before the `next` function, you get the original state. All middleware packages for Redux are based on these basic principles.

In the following sections, we'll introduce you to Redux Thunk, Redux Saga, and Redux Observable, three of the most popular async middleware packages for Redux, which you can use to connect your Redux application to the server.

15.2 Redux with Redux Thunk

The application from the previous chapter serves as the basis for the examples in this chapter. You created this application using the `redux-typescript` template from Create React App. The application consists of a slice that helps you to manage a list of books. Currently, the application only works locally in the browser. This means that any changes you make will be lost during a reload process.

To link your application's interface to the server, you need four asynchronous operations:

- Reading of existing data records
- Creation of new data records
- Modification of existing data records
- Deletion of data records

You'll solve this task in the following sections using all three libraries. In the first step, you'll use *Redux Thunk*. This is the most obvious variant, as it is a fixed part of Redux Toolkit and therefore you don't need to install any additional packages or configure Redux further. Moreover, Redux Thunk is the simplest and lightest variant of asynchronous middleware. This middleware is based, as the name suggests, on so-called thunks. These are functions that you use to perform calculations after the current execution flow. In the actual implementation of Redux Thunk, a *thunk* is nothing more than a function returned by an action creator. This function has access to the `dispatch` and `getState` functions of the store. In the thunk function, you then have the option to perform your asynchronous operation and use the result to dispatch an action in turn, which is then picked up and handled by the reducer.

15.2.1 Manual Integration of Redux Thunk

Redux Thunk is a standalone library that's being developed in parallel with Redux. The project's repository can be found at <https://github.com/reduxjs/redux-thunk>. If you don't use Redux Toolkit, you can install Redux Thunk in your application using the `npm install redux-thunk` command. In this case, you must integrate the middleware yourself. In [Listing 15.2](#), you can see how the integration works:

```
import { createStore, applyMiddleware } from 'redux';
import rootReducer from '../reducers/rootReducer';
import thunk from 'redux-thunk';

export function configureStore() {
  return createStore(rootReducer, applyMiddleware(thunk));
}
```

Listing 15.2 Integration of Redux Thunk into the Application

For manual integration, you use the `applyMiddleware` function of Redux, which allows you to integrate middleware into your application, and you pass the thunk middleware to this function. This completes the setup process, and you can move on to implementing asynchronous operations.

As mentioned, you don't have to bother with installing and integrating Redux Thunk yourself if you use Redux Toolkit. On the following pages, we assume that you're using Redux Toolkit because it means that there is much less work for you.

15.2.2 Reading Data from the Server

The main focus of Redux Thunk is on the actions or action creator functions. With the integration of this library, the basic principle of *action creators* changes in Redux. Normally, such a function returns a simple JavaScript object that follows a certain structure. With Redux Thunk, action creators can return not only simple objects, but also functions. These functions contain logic that causes the dispatching of actions. They take care of the asynchronous operations, such as server communication, and in turn dispatch actions themselves.

Preparation: The Server Interface

As a server interface, you use the `json-server` package, as you have elsewhere in this book, which provides a lightweight backend. To make the backend work, you first install the package using the `npm install json-server` command and create a file named `data.json` in the root directory. The contents of this file are shown in [Listing 15.3](#):

```
{
  "books": [
    {
      "id": 1,
      "title": "JavaScript - The Comprehensive Guide",
      "author": "Philip Ackermann",
      "isbn": "978-3836286299",
      "rating": 5
    },
    {
      "id": 2,
      "title": "Clean Code",
      "author": "Robert Martin",
      "isbn": "978-0132350884",
      "rating": 4
    },
    {
      "id": 3
      "title": "Design Patterns",
```

```

    "author": "Erich Gamma",
    "isbn": "978-0201633610",
    "rating": 5
  }
]
}

```

Listing 15.3 Data Source for the Backend (package.json)

With these structures, you can start your backend using the `npx json-server -p 3001 -w data.json` command. It works a bit more comfortably if you add an entry to the `scripts` section of your `package.json` file. You can see the corresponding section in [Listing 15.4](#):

```

{
  ...
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test",
    "eject": "react-scripts eject",
    "backend": "json-server -p 3001 -w data.json"
  },
  ...
}

```

Listing 15.4 package.json Script for the Backend

This extension enables you to start your backend via `npm run backend`. Once you've started your backend, you need to make sure that the backend is running in the background and in parallel with the frontend.

Thunk to Read the Data from the Server

The first step toward asynchronicity in Redux with Redux Toolkit is to create a Redux Thunk action creator using the `createAsyncThunk` function. The function accepts a character string representing the type of the action and a function encapsulating the asynchronous operation. You call this function in the `booksSlice.ts` file in the `src/features/books` directory and export the return value as `loadData` constants. You can see the corresponding source code in [Listing 15.5](#):

```

import {
  createAsyncThunk,
  createSelector,
  createSlice,
  PayloadAction,

```

```
} from '@reduxjs/toolkit';
import { RootState } from '../app/store';
import { Book, InputBook } from './Book';
import booksData from './booksData';

export type BooksState = {...};

export const loadData = createAsyncThunk(
  'books/loadData',
  async (obj, { rejectWithValue }) => {
    try {
      const response = await fetch('http://localhost:3001/books');
      if (response.ok) {
        const data = await response.json();
        return data;
      }
      return Promise.reject();
    } catch (e) {
      return rejectWithValue(e);
    }
  }
);
export const booksSlice = createSlice({...});
...
```

Listing 15.5 Creating a Redux Thunk Action Creator Using the “createAsyncThunk” Function (src/features/books/booksSlice.ts)

Calling the `createAsyncThunk` function creates three action types: `books/loadData/pending`, `books/loadData/fulfilled`, and `books/loadData/rejected`. These three types replicate the lifecycle of the asynchronous operation and allow you to respond to the different phases. The callback function you pass to the `createAsyncThunk` function receives the data you passed when dispatching the thunk, and the Thunk API as the second argument. For example, here you get access to the `rejectWithValue` function, which allows you to pass additional information to the reducer that handles the operation failure.

The server request may fail for two different reasons: either the `fetch` function throws an exception or the `ok` property of the `response` object contains the value `false`. In both cases, you must ensure that the thunk function triggers the `rejected` action. You can do this either by returning a rejected promise or by using the `rejectWithValue` function from Redux Toolkit. If you call the `rejectWithValue` function with a value, Redux Toolkit ensures that the value is passed as the payload of the action. If your reducer doesn't work with a payload, you can also return a rejected `Promise` object.

As you know, asynchronous operations and the modification of the store are decoupled in Redux. This means that you also have to extend your reducer to be able to respond to the respective action types. You also make this modification in the *booksSlice.ts* file, as shown in [Listing 15.6](#):

```
import {
  createAsyncThunk,
  createSelector,
  createSlice,
  PayloadAction,
} from '@reduxjs/toolkit';
import { RootState } from '../app/store';
import { Book, InputBook } from './Book';

export type BooksState = {
  books: Book[];
  loadingState: null | 'pending' | 'completed' | 'error';
  ratingFilter: number;
};

export const loadData = createAsyncThunk('books/
loadData', async (obj, { rejectWithValue }) => { ...});

export const booksSlice = createSlice({
  name: 'books',
  initialState: { books: [], loadingState: null, ratingFilter: 0 } ←
    as BooksState,
  reducers: {...},
  extraReducers: (builder) => {
    builder
      .addCase(loadData.pending, (state) => {
        state.loadingState = 'pending';
      })
      .addCase(loadData.fulfilled, (state, action) => {
        state.loadingState = 'completed';
        state.books = action.payload;
      })
      .addCase(loadData.rejected, (state) => {
        state.loadingState = 'error';
      });
  },
});

export const { remove, save } = booksSlice.actions;
```



```
export const selectBooks = (state: RootState) => state.books.books;
export const selectLoadingState = (state: RootState) =>
  state.books.loadingState;
export const selectRatingFilter = (state: RootState) =>
  state.books.ratingFilter;

export const selectByRating = createSelector(...);

export function selectBook(state: RootState): (id?: number) => InputBook {...}

export default booksSlice.reducer;
```

Listing 15.6 Integration of the Thunk into the Slice (src/features/books/booksSlice.ts)

You should make sure that your users are always informed about what's happening in your application. This means that you should also display the state of the asynchronous operation in the graphical interface. For this reason, you extend the `BooksState` structure with the `loadingState` property. This property can have four possible values: `null`, `pending`, `completed`, and `error`. The `null` value represents the initial value before you start the server communication. Then `loadingState` changes to `pending` and finally to `completed` or `error`.

In the slice itself, you adjust the `initialState` structure and start there with an empty array for displaying the data and a `loadingState` with the value `null`. To make sure the typing process works correctly, you can either take advantage of the fact that the `createSlice` function is implemented as a generic function to which you can pass the state structure and reducers, or you can use the `as` keyword in `initialState` to specify the desired structure so that TypeScript uses the appropriate types by means of type inference. The second variant reduces your work, so we'll choose it for our example.

This structure enables you to dispatch the thunk action and trigger the server communication with it, but the actual work is done by the functions in the `extraReducers` property. The `builder` object allows you to add cases for the actions, which is similar to what the reducer does. In total, you create three cases, for the `pending`, `fulfilled`, and `rejected` states of the asynchronous operation. Within these functions, you have access to the state of the slice and the triggered action. You can use this information to create a new version of the state and adjust the `loadingState` property and, if necessary, the `books` property accordingly.

To access the `loadingState` from your components, you still need to define a selector function, and you implement it in a way similar to the `selectBooks` function.

Your application still works completely without a server, although it doesn't display any data in the meantime. To solve this problem, you need to trigger the asynchronous operation.

Integration of the Asynchronous Operation into the “List” Component

The List component in the `src/features/books` directory is responsible for displaying the data and is therefore also a suitable place where you can take care of loading the data from the server. Thus, when mounting the component, you trigger the `loadData` action and make sure that the component also reflects the state of the asynchronous operation. In [Listing 15.7](#), you can see the source code of this component:

```
import React, { useEffect } from 'react';
import { useSelector } from 'react-redux';
import { useNavigate } from 'react-router-dom';
import { useAppDispatch } from '../../app/hooks';
import {
  selectBooks,
  remove,
  selectLoadingState,
  loadData,
} from './booksSlice';

const List: React.FC = () => {
  const books = useSelector(selectBooks);
  const loadingState = useSelector(selectLoadingState);
  const dispatch = useAppDispatch();
  const navigate = useNavigate();

  useEffect(() => {
    dispatch(loadData());
  }, [dispatch]);

  if (loadingState === 'pending') {
    return <div>...loading</div>;
  } else if (loadingState === 'error') {
    return <div>An error has occurred!</div>;
  } else {
    return (
      <>
        <table>...</table>
        <button onClick={() => navigate('/new')}>new</button>
      </>
    );
  }
};

export default List;
```

Listing 15.7 Connection of the “List” Component with Redux Thunk
(`src/features/books/List.tsx`)

You use the `selectLoadingState` selector to get the information about the state of the asynchronous request. This is needed to inform your users of what's currently happening in the background. You trigger the asynchronous operation by executing the `loadData` function from `booksSlice` and passing the result to Redux via the `dispatch` function.

This means that React initially renders the component with an empty `books` array and a `loadingState` with the value `null`. Then the effect hook takes effect and triggers the asynchronous operation. The `books` array is still empty, but the `loadingState` has the value `pending`. The component then renders a `div` element with `...loading`. Once the asynchronous operation is finished, the component displays either an error message or the list of book records. So normally you should see `...loading` very briefly, followed immediately by the book list.

If you want to test the error handling, you merely need to change the URL in `booksSlice` so that it becomes invalid. As a result, the server responds with status code 404 and the component renders the error message. [Figure 15.1](#) shows the list display with the opened Redux Dev Tools.

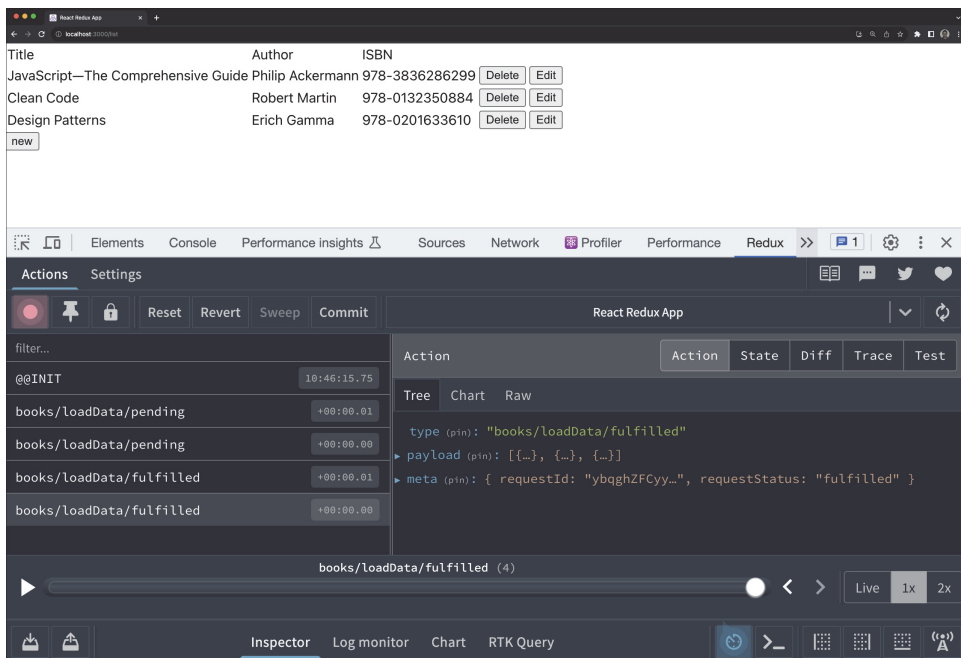


Figure 15.1 Display of the List

As you can see, the `pending` and `fulfilled` actions are triggered twice. However, there's no reason for concern here. This is due to the `StrictMode` component of React in development mode. You can disable the `StrictMode` component in the `index.tsx` file, and then you'll see the actions only once. You should still enable `StrictMode`, as it can give you

valuable clues about potential problems. In the production mode of your application, `StrictMode` is deactivated and the actions as well as the server communication take place only once.

Your Redux application is now able to receive data from the server and to handle errors too. In the next step, we turn to write operations, starting with deleting data records.

15.2.3 Deleting Data Records

In the previous chapter, you implemented the deletion of data records locally by means of a `remove` action and the associated reducer. The `List` component contains a button for each data record that allows you to delete it. Clicking the button dispatches the `remove` action.

You don't need to make any adjustments in your component to integrate Redux Thunk. You only need to integrate the newly added error-handling functionality. But we'll return to that later. Let's first turn our attention to `booksSlice`. Here you extend the state structure with an additional state for the delete operation, create a thunk to delete the data records, and integrate the three actions into the `extraReducers` property of the slice:

```
import {
  createAsyncThunk,
  createSelector,
  createSlice,
  PayloadAction,
} from '@reduxjs/toolkit';
import { RootState } from '../app/store';
import { Book, InputBook } from './Book';

export type BooksState = {
  books: Book[];
  loadingState: null | 'pending' | 'completed' | 'error';
  removeState: null | 'pending' | 'completed' | 'error';
  ratingFilter: number;
};

export const loadData = createAsyncThunk(...);

export const remove = createAsyncThunk(
  'books/remove',
  async (id: number, { rejectWithValue }) => {
    try {
      const response = await fetch(`http://localhost:3001/books/${id}`, {
        method: 'DELETE',
      });
    } catch (error) {
      return rejectWithValue(error);
    }
  }
);
```

```
    });
    if (response.ok) {
      return id;
    }
    return Promise.reject();
  } catch (e) {
    return rejectWithValue(e);
  }
}
);

export const booksSlice = createSlice({
  name: 'books',
  initialState: {
    books: [],
    loadingState: null,
    removeState: null,
    ratingFilter: 0,
  } as BooksState,
  reducers: {
    save(state, action: PayloadAction<InputBook>) {...},
  },
  extraReducers: (builder) => {
    builder
      .addCase(loadData.pending, (state) => {
        state.loadingState = 'pending';
      })
      .addCase(loadData.fulfilled, (state, action) => {
        state.loadingState = 'completed';
        state.books = action.payload;
      })
      .addCase(loadData.rejected, (state) => {
        state.loadingState = 'error';
      });

    builder
      .addCase(remove.pending, (state) => {
        state.removeState = 'pending';
      })
      .addCase(remove.fulfilled, (state, action) => {
        state.removeState = 'completed';
        const index = state.books.findIndex(
          (book) => book.id === action.payload
        );
      });
  }
});
```

```

        state.books.splice(index, 1);
      })
      .addCase(remove.rejected, (state) => {
        state.removeState = 'error';
      });
    },
  });

export const { save } = booksSlice.actions;

export const selectBooks = (state: RootState) => state.books.books;
export const selectLoadingState = (state: RootState) =>
  state.books.loadingState;
export const selectRemoveState = (state: RootState) =>
  state.books.removeState; ←
export const selectRatingFilter = (state: RootState) =>
  state.books.ratingFilter;

export const selectByRating = createSelector(...);

export function selectBook(state: RootState): (id?: number) => InputBook {...}

export default booksSlice.reducer;

```

Listing 15.8 Integration of the Deletion Routine into “BooksSlice”
(src/features/books/booksSlice.ts)

Here, `removeState` in the `BooksState` structure has the same structure as `loadingState`. Initially, you set the value to `null`. It then assumes the value `pending`, `completed`, or `error`.

Then you name the thunk `books/remove` and implement a `fetch` call in the callback function with the method property `DELETE` and the same error handling routine as when you read the data. In `extraReducers`, you create a new block with the case handlings for the `remove` actions. There you have the option to specify all cases in one block or to split them into several blocks, as in the example. Such a separation makes reading the source code much easier. The functionality is the same no matter what approach you take.

To access `removeState`, you define the `selectRemoveState` selector.

This adjustment of the slice allows you to move onto integrating the functionality into the `List` component in the next step. Here, in addition to dispatching the `remove` action from the previous chapter, you only need to worry about processing the `removeState`. Although the integration of the asynchronous thunk has changed the internal structure of the action creator function, the name and signature haven’t changed, so you

don't need to worry about anything else here. In [Listing 15.9](#), you can see the source code of the `List` component:

```
import React, { useEffect } from 'react';
...

const List: React.FC = () => {
  const books = useSelector(selectBooks);
  const loadingState = useSelector(selectLoadingState);
  const removeState = useSelector(selectRemoveState);
  const dispatch = useAppDispatch();
  const navigate = useNavigate();

  useEffect(() => {
    dispatch(loadData());
  }, [dispatch]);

  if (loadingState === 'pending') {
    return <div>...loading</div>;
  } else if (loadingState === 'error') {
    return <div>An error has occurred!</div>;
  } else {
    return (
      <>
        {removeState === 'pending' && <div>Data record will be ←
          deleted</div>}
        {removeState === 'error' && (
          <div>An error occurred during deletion</div>
        )}
      <table>
        <thead>...</thead>
        <tbody>
          {books.map((book) => (
            <tr key={book.id}>
              ...
              <td>
                <button onClick={() => dispatch(remove(book.id))}>
                  delete
                </button>
              </td>
              ...
            </tr>
          ))}
        </tbody>
      </table>
    )
  }
}
```

```

    </table>
    <button onClick={() => navigate('/new')}>new</button>
  </>
);
}
};

export default List;

```

Listing 15.9 Integration of the Thunk for Deleting Data Records (src/features/books/List.tsx)

To test the changes, you need to make sure that the backend process runs in parallel with the build process of the frontend. Then you can switch to the browser and delete records. During the local execution, you usually don't get to know anything about the pending state. But you can change that with a little trick: in Chrome's developer tools, you have the option to configure the bandwidth. This option is referred to as *throttling*. For example, if you select **Throttling • Slow 3G** in the developer tools, the connection to the server will be slowed down significantly, so you should see the “Data record will be deleted” message at least briefly. You can provoke an error by either terminating the backend process or specifying an invalid URL for the backend query. In both cases, the List component will report that an error has occurred.

You can also track which actions are triggered when a data record is deleted in Redux Dev Tools, as shown in [Figure 15.2](#).

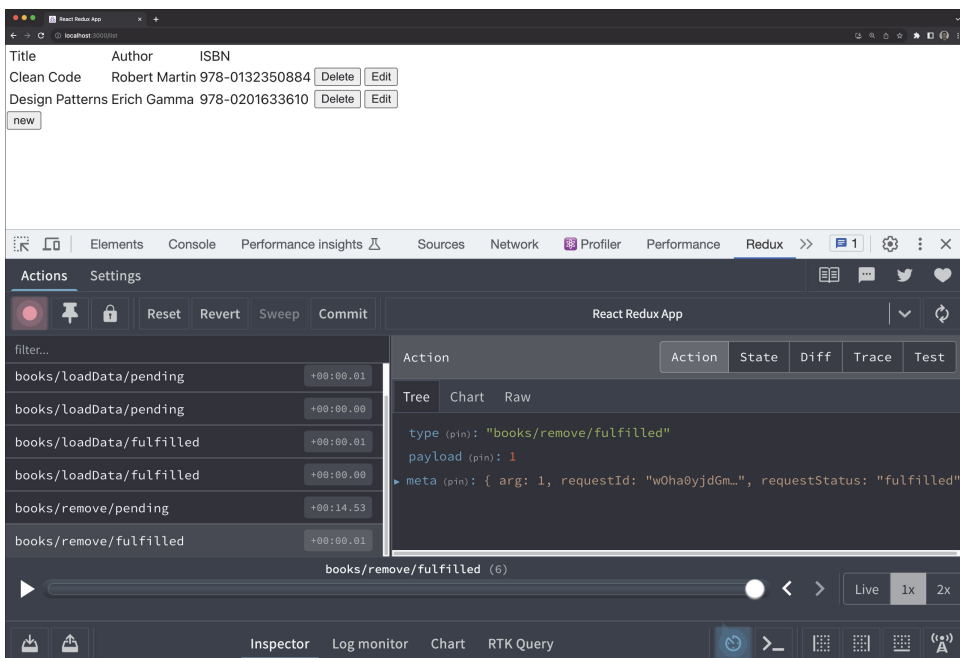


Figure 15.2 Deleting the Data Records in the Redux Dev Tools

15.2.4 Creating and Modifying Data Records

The creation and modification of data records follows the same schema as the deletion. Again, for the most part, you need to adjust the implementation of the slice. You can either handle creating and modifying separately, or you can combine the two actions. If you handle them separately, you need to implement two different thunks and dispatch the appropriate action in the component—in this case, the `Form` component. This has the advantage that the implementations are cleanly separated from each other, while the drawback is that the component must know whether it's a new data record and what it has to do in this case. We therefore prefer the variant where you dispatch a save action, which then takes care of either creating the data record or modifying it. We'll follow this strategy in the following example.

In `BooksSlice`, you create a new thunk for storing the data on the server. In the callback function, you decide whether the record is new or existing based on the presence of the `id` property and configure the request accordingly. The error handling you operate as you already did with `read` and `delete` and mark the operation as having failed. To access the state of the asynchronous operation, you implement an additional selector function. [Listing 15.10](#) contains the updated `BooksSlice`:

```
import {
  createAsyncThunk,
  createSelector,
  createSlice,
} from '@reduxjs/toolkit';
import { RootState } from '../app/store';
import { Book, InputBook } from './Book';

export type BooksState = {
  books: Book[];
  loadingState: null | 'pending' | 'completed' | 'error';
  removeState: null | 'pending' | 'completed' | 'error';
  saveState: null | 'pending' | 'completed' | 'error';
  ratingFilter: number;
};

export const loadData = createAsyncThunk(...);

export const remove = createAsyncThunk(...);

export const save = createAsyncThunk(
  'books/save',
  async (book: InputBook, { rejectWithValue }) => {
    try {
      let url = 'http://localhost:3001/books';
```

```

    let method = 'POST';
    if (book.id) {
      url += `/${book.id}`;
      method = 'PUT';
    }

    const response = await fetch(url, {
      method,
      headers: {
        'Content-Type': 'application/json',
      },
      body: JSON.stringify(book),
    });
    if (response.ok) {
      const data = await response.json();
      return data;
    }
    return Promise.reject();
  } catch (e) {
    return rejectWithValue(e);
  }
}
);

export const booksSlice = createSlice({
  name: 'books',
  initialState: {
    books: [],
    loadingState: null,
    removeState: null,
    saveState: null,
    ratingFilter: 0,
  } as BooksState,
  reducers: {},
  extraReducers: (builder) => {
    ...

    builder
      .addCase(save.pending, (state) => {
        state.saveState = 'pending';
      })
      .addCase(save.fulfilled, (state, action) => {
        if (action.payload.id) {
          const index = state.books.findIndex(

```

```

        (book) => book.id === action.payload.id
      );
      state.books[index] = action.payload as Book;
    } else {
      state.books.push(action.payload);
    }
  })
  .addCase(save.rejected, (state) => {
    state.saveState = 'error';
  });
},
});

export const selectBooks = (state: RootState) => state.books.books;
export const selectLoadingState = (state: RootState) =>
  state.books.loadingState;
export const selectRemoveState = (state: RootState) =>
  state.books.removeState;
export const selectSaveState = (state: RootState) =>
  state.books.saveState;
export const selectRatingFilter = (state: RootState) =>
  state.books.ratingFilter;

export const selectByRating = createSelector(...);

export function selectBook(state: RootState): (id?: number) => InputBook {...}

export default booksSlice.reducer;

```

Listing 15.10 Saving Data Records in “BooksSlice” (src/features/books/booksSlice.ts)

The next step is to connect the new interfaces from the slice with the `Form` component. Again, all you need to do is integrate additional information about the operation's progress as the action you're dispatching to save has only changed internally.

```

import React, { useEffect } from 'react';
import { useSelector } from 'react-redux';
import { save, selectBook, selectSaveState } from './booksSlice';
import { useForm } from 'react-hook-form';
import { useNavigate, useParams } from 'react-router-dom';
import { useAppDispatch } from '../../app/hooks';
import { InputBook } from './Book';

const Form: React.FC = () => {
  const getBook = useSelector(selectBook);

```

```

const dispatch = useAppDispatch();
const navigate = useNavigate();
const { register, handleSubmit, reset } = useForm<InputBook>({
  defaultValues: { title: '', author: '', isbn: '' },
});
const saveState = useSelector(selectSaveState);

const { id } = useParams<{ id?: string }>();

useEffect(() => {
  if (id) {
    const book = getBook(parseInt(id, 10));
    reset(book);
  }
}, [id, reset, getBook]);

return (
  <>
    {saveState === 'pending' && <div>Data will be saved.</div>}
    {saveState === 'error' && <div>An error has occurred.</div>}

    <form...>...</form>
  </>
);
};

export default Form;

```

Listing 15.11 Integration of the “save” Thunk into the “Form” Component
(src/features/books/Form.tsx)

With this adjustment, you can now manage the data records in your application. You can view, modify, and delete existing records and create new ones.

Redux Thunk is one of the simplest variants of asynchronous middleware for Redux. In the following sections, you’ll learn how to use Redux Saga to add an even more versatile tool to your application.

15.3 Generators: Redux Saga

The basic principle of *Redux Saga* is similar to that of Redux Thunk: you use the middleware to intercept actions that have a side effect, such as the communication with a web server. Unlike Redux Thunk, the action that triggers the operation is also triggered by a

regular action. The reducer ignores this action and returns the original state unchanged. The saga function executes the asynchronous operation and, depending on the result, triggers another action that is then handled by the reducer.

As with Thunk, the term *saga* does not represent a Redux-specific operation. A saga is a general design pattern in development and ensures that there is appropriate error handling for asynchronous operations that run longer. In essence, the design pattern says that for each workflow step in the application, there should be a countermeasure that, in the event of an error, ensures that the action can be rolled back.

In the actual implementation of Redux Saga, ECMAScript generators are used to keep the asynchronous code clear and easy to read.

ECMAScript Generators

Generators are objects that reflect a suspended application state. A generator object is created using a generator function. Instead of using the usual `return` statements, in a generator function you return the values with the `yield` keyword:

```
function *generatorFunc() {
  for(let i = 0; i < 2; i++) {
    yield i;
  }
}

const generator = generatorFunc();
console.log(generator.next()); // { value: 0, done: false }
console.log(generator.next()); // { value: 1, done: false }
console.log(generator.next()); // { value: undefined, done: true }
```

Listing 15.12 Generator Function in JavaScript

Listing 15.12 shows a simple example of a generator function. This type of function is marked with an asterisk. Inside such a function, the `yield` keyword is allowed. You get the actual generator object by calling the function. The generator object has the `next` method, to which you can optionally pass a value in order to influence the further flow of the generator. As return of the `next` method, you get an object with the `value` and `done` keys. The `value` property contains the value you return with `yield`. The `done` property has a Boolean value, and `true` stands for the fact that the generator is completed and produces no further values. In this case, the `value` property is assigned the value `undefined`. If the `done` property has the value `false`, then this indicates that there are other values in the generator.

You can use the generator object not only directly, but also in a `for ... of` loop. In this case, the program iterates over all possible values and terminates the loop as soon as the `done` property has the value `true`.

A special feature of generator functions is that the function is paused between two calls of `next`.

15.3.1 Installation and Integration of Redux Saga

Redux Saga is an open-source project developed on GitHub. It's a standalone project based on Redux, but developed independently. The Redux Saga project is managed as a monorepo and already includes appropriate type definitions. You need to use the `npm install redux-saga` command to install the package in your application.

Being Redux middleware, Redux Saga must be integrated when the store is created by means of the `middleware` method. You return an array with the standard middleware and the Saga middleware as the value. You get the default middleware as the return value of the `getDefaultMiddleware` function, which you get as an argument of the `middleware` method. After that, the sagas have access to all actions dispatched in the application. The integration takes place in three stages: first, you create the saga middleware; second, you integrate it into the store; and in the last step, you integrate the sagas of your application. In [Listing 15.13](#), you can see the implementation of these three steps in the `store.ts` file:

```
import saga from 'redux-saga';
import { Action, configureStore, ThunkAction } from '@reduxjs/toolkit';
import counterReducer from '../features/counter/counterSlice';
import booksReducer from '../features/books/booksSlice';
import rootSaga from './RootSaga';

const sagaMiddleware = saga();

export const store = configureStore({
  reducer: {
    counter: counterReducer,
    books: booksReducer,
  },
  devTools: true,
  middleware(getDefaultMiddleware) {
    return [...getDefaultMiddleware(), sagaMiddleware];
  },
});

sagaMiddleware.run(rootSaga);

export type AppDispatch = typeof store.dispatch;
export type RootState = ReturnType<typeof store.getState>;
export type AppThunk<ReturnType = void> = ThunkAction<
  ReturnType,
  RootState,
  unknown,
  Action<string>
>;
```

Listing 15.13 Integration of Redux Saga into the Application (`src/app/store.ts`)

You use the `saga` function to create a middleware instance that you can integrate into your store using the `middleware` method. In the final step, you run the `run` method of the middleware and pass the `rootSaga` function, which you'll implement in the next step.

15.3.2 Loading Data from the Server

The first operation you implement with Redux Saga concerns reading data from the server. For this purpose, you first need to create actions in the `books.actions.ts` file in the `src/features/books` directory. Unlike Redux Thunk, the action creator function doesn't contain any logic. This means that you can either create the action objects yourself, use the `createAction` function of the Redux Toolkit, or rely on additional packages such as `typesafe-actions`.

In the example, we use the `typesafe-actions` package, which you can install via the `npm install typesafe-actions` command. The source code of the `books.actions.ts` file, which contains the call of `createAsyncAction`, is shown in [Listing 15.14](#):

```
import { createAsyncAction } from 'typesafe-actions';
import { Book } from './Book';

export const loadDataAction = createAsyncAction(
  'books/loadData/pending',
  'books/loadData/fulfilled',
  'books/loadData/rejected'
)<void, Book[], void>();
```

Listing 15.14 Actions to Load the Data (`src/features/books/books.actions.ts`)

In the `books.actions.ts` file, you call the `createAsyncAction` function with three strings that represent the type values of the three action objects this function creates. Next, you specify the types of action payloads.

Due to the strict separation between state management and display, the modifications are limited to the Redux-specific parts of the application. The components remain unaffected. The core elements of Redux Saga are the so-called sagas; they take care of the task that the thunks handled in Redux Thunk, but in a saga you specify to which action you want to respond. You then execute the desired asynchronous operation and dispatch another action yourself, which is then handled by the reducer. [Listing 15.15](#) contains the source code of the Books saga, which you store in the `books.saga.ts` file in the `src/features/books` directory:

```
import { takeLatest, put } from '@redux-saga/core/effects';
import { Book } from './Book';
import { loadDataAction } from './books.actions';
```

```
function* loadData(): Generator {
  try {
    const response: Response = (yield fetch(
      'http://localhost:3001/books'
    )) as Response;
    if (response.ok) {
      const data = (yield response.json()) as Book[];
      yield put(loadDataAction.success(data));
    } else {
      yield put(loadDataAction.failure());
    }
  } catch (e) {
    yield put(loadDataAction.failure());
  }
}

export default function* booksSaga() {
  yield takeLatest(loadDataAction.request, loadData);
}
```

Listing 15.15 “Books” Saga Implementation (src/features/books/books.saga.ts)

The entry point to the Books saga is the `booksSaga` function, which is implemented as a generator function. The `takeLatest` function allows you to specify that the `loadData` function should be executed if the `loadDataAction.request` action has been dispatched. The special feature of the `takeLatest` function is that it ensures that only one parallel request for the data is sent to the server. If a request is already executed and is to be requested again, the first request is canceled and only the newer request will be processed. As an alternative, you can use the `takeEvery` function. In this case, all requests will be processed and none of them will get canceled.

The `loadData` function takes care of loading resources from the server. Because this function is also a generator function, you use `yield` instead of the `await` keyword. Using the `try ... catch` statement, you make sure that the errors that occur during the communication with the server are caught as well. The `put` function of Redux Saga dispatches the appropriate action. In the case of a success, you call the `loadDataAction.success` method, whereas in the case of a failure, you use the `loadDataAction.failure` method.

Now that you’ve implemented your saga, you still need to integrate it in your application. This step takes place in the root saga. The root saga enables you to collect and register all sagas of the application. You place this function in the `rootSaga.ts` file in the `src/app` directory. The source code of this file is shown in [Listing 15.16](#):

```
import { all } from '@redux-saga/core/effects';
import booksSaga from '../features/books/books.saga';
```



```
export default function* rootSaga() {  
  yield all([booksSaga()]);  
}
```

Listing 15.16 Implementation of the Root Saga (src/sagas/rootSaga.ts)

The core of the root saga is the `all` function from the `redux-saga` package. You can pass an array of saga function calls to it, which will then be registered. But there are also other possibilities—for example, the use of the `fork` function, to which you pass the saga functions and which behaves similarly to the `all` function.

In the final step, you clean up `BooksSlice`, remove the thunk to load the data, and trigger the saga instead. In the `booksSlice.ts` file, you remove the `loadData` constant and the associated `createAsyncThunk` call. Also, in `extraReducers`, you adjust the `addCase` calls for `loadData`, as shown in [Listing 15.17](#):

```
import {  
  createAsyncThunk,  
  createSelector,  
  createSlice,  
} from '@reduxjs/toolkit';  
import { ActionType, getType } from 'typesafe-actions';  
import { RootState } from '../app/store';  
import { Book, InputBook } from './Book';  
import { loadDataAction } from './books.actions';  
  
export type BooksState = {...};  
  
export const remove = createAsyncThunk(...);  
  
export const save = createAsyncThunk(...);  
  
export const booksSlice = createSlice({  
  name: 'books',  
  initialState: {...} as BooksState,  
  reducers: {},  
  extraReducers: (builder) => {  
    builder  
      .addCase(getType(loadDataAction.request), (state) => {  
        state.loadingState = 'pending';  
      })  
      .addCase(  
        getType(loadDataAction.success),  
        (state, action: ActionType<typeof loadDataAction.success>) => {  
          state.loadingState = 'completed';  
          state.books = action.payload;  
        }  
      )  
  }  
});
```

```

    )
    .addCase(getType(loadDataAction.failure), (state) => {
      state.loadingState = 'error';
    });

    ...
  },
});

export const selectBooks = (state: RootState) => state.books.books;
export const selectLoadingState = (state: RootState) =>
  state.books.loadingState;
export const selectRemoveState = (state: RootState) =>
  state.books.removeState;
export const selectSaveState = (state: RootState) =>
  state.books.saveState;
export const selectRatingFilter = (state: RootState) =>
  state.books.ratingFilter;

export const selectByRating = createSelector(...);

export function selectBook(state: RootState): (id?: number) => InputBook {...}

export default booksSlice.reducer;

```

Listing 15.17 Adjustments to “BooksSlice” (src/features/books/booksSlice.ts)

The integration of Redux Saga has also created some problems in the components. In the Form component, you cast the output of the save function that creates the action object for saving to AnyAction. You can achieve this via the save(event) as unknown as AnyAction statement. The AnyAction type comes from the @reduxjs/toolkit package. This cast is just an intermediate step that you will remove later.

In the List component, you also need to make an adjustment and cast the remove action as well, and for loading the data, you include the newly created action creator function.

Listing 15.18 contains the source code of the List component:

```

import { AnyAction } from '@reduxjs/toolkit';
import React, { useEffect } from 'react';
import { useSelector } from 'react-redux';
import { useNavigate } from 'react-router-dom';
import { useAppDispatch } from '../../app/hooks';
import { loadDataAction } from '../books.actions';
import {
  selectBooks,
  remove,
  selectLoadingState,

```

```

    selectRemoveState,
  } from './booksSlice';

const List: React.FC = () => {
  const books = useSelector(selectBooks);
  const loadingState = useSelector(selectLoadingState);
  const removeState = useSelector(selectRemoveState);
  const dispatch = useAppDispatch();
  const navigate = useNavigate();

  useEffect(() => {
    dispatch(loadDataAction.request());
  }, [dispatch]);

  if (loadingState === 'pending') {
    return <div>...loading</div>;
  } else if (loadingState === 'error') {
    return <div>An error has occurred!</div>;
  } else {
    return (
      <>
        {removeState === 'pending' && <div>Data record will be ↩
          deleted</div>}
        {removeState === 'error' && (
          <div>An error occurred during deletion</div>
        )}
      <table>
        <thead>
          <tr>
            <td>Title</td>
            <td>Author</td>
            <td>ISBN</td>
            <td></td>
            <td></td>
          </tr>
        </thead>
        <tbody>
          {books.map((book) => (
            <tr key={book.id}>
              <td>{book.title}</td>
              <td>{book.author}</td>
              <td>{book.isbn}</td>
              <td>
                <button
                  onClick={() =>
                    dispatch(remove(book.id) as unknown as AnyAction)
                  }

```


In the next step, you implement the deletion of data records.

15.3.3 Deleting Existing Data

When deleting data records, you proceed similarly to the implementation of the loading routine: You create the necessary action creator functions that ensure that the request gets sent to the server and the response will be handled. You also add a server communication feature to the existing saga. The first step consists of extending the *books.actions.ts* file, as shown in [Listing 15.19](#):

```
import { createAsyncAction } from 'typesafe-actions';
import { Book } from '../Book';

export const loadDataAction = createAsyncAction(...)<void, Book[], void>();

export const removeAction = createAsyncAction(
  'books/remove/pending',
  'books/remove/fulfilled',
  'books/remove/rejected'
)<number, number, void>();
```

Listing 15.19 Creating “removeAction” (*src/features/books/books.actions.ts*)

In the next step, you extend the *books.saga.ts* file with a generator function called *remove*. This way you ensure that the delete request is sent to the server and dispatches either the *removeAction.success* or the *removeAction.failure* action, depending on the result of this request. You include this generator function in the *booksSaga* function, just as you did with the *loadData* function. The corresponding source code is shown in [Listing 15.20](#):

```
import { takeLatest, put } from '@redux-saga/core/effects';
import { Book } from '../Book';
import { loadDataAction, removeAction } from '../books.actions';

function* loadData(): Generator {...}

function* remove({
  payload: id,
}: ReturnType<typeof removeAction.request>): Generator {
  try {
    const response = (yield fetch(`http://localhost:3001/books/${id}`, {
      method: 'DELETE',
    })) as Response;
    if (response.ok) {
      yield put(removeAction.success(id));
    }
  }
}
```

```

    } else {
      yield put(removeAction.failure());
    }
  } catch (e) {
    yield put(removeAction.failure());
  }
}

export default function* booksSaga() {
  yield takeLatest(loadDataAction.request, loadData);
  yield takeLatest(removeAction.request, remove);
}

```

Listing 15.20 Implementation of the “remove” Saga (*src/features/books/books.saga.ts*)

When you trigger the `removeAction.request` action, you pass the ID of the data record to be deleted. This ID is then available to you in the `remove` saga. Along with the ID, you send a DELETE request to the server and check if the response was OK. If that’s not the case or an exception was thrown, you use the `put` function of Redux Saga to dispatch the `removeAction.failure` action. If successful, you trigger the `removeAction.success` action and pass the ID of the deleted data record.

Because you’ve already integrated the `remove` saga into `booksSaga`, you don’t need to do anything else here.

With the switch to Redux Saga, you’ll have to adjust `BooksSlice` a little, delete the `thunk` function, and switch to the new actions. You can see how this affects the source code of the *booksSlice.ts* file in [Listing 15.21](#):

```

import {
  createAsyncThunk,
  createSelector,
  createSlice,
} from '@reduxjs/toolkit';
import { ActionType, getType } from 'typesafe-actions';
import { RootState } from '../app/store';
import { Book, InputBook } from '../Book';
import { loadDataAction, removeAction } from '../books.actions';

export type BooksState = {...};

export const save = createAsyncThunk(...);

export const booksSlice = createSlice({
  name: 'books',
  initialState: {...} as BooksState,

```

```
reducers: {},
extraReducers: (builder) => {
  builder
    .addCase(getType(loadDataAction.request), (state) => {
      ...
    });

  builder
    .addCase(getType(removeAction.request), (state) => {
      state.removeState = 'pending';
    })
    .addCase(
      getType(removeAction.success),
      (state, action: ActionType<typeof removeAction.success>) => {
        state.removeState = 'completed';
        const index = state.books.findIndex(
          (book) => book.id === action.payload
        );
        state.books.splice(index, 1);
      }
    )
    .addCase(getType(removeAction.failure), (state) => {
      state.removeState = 'error';
    });

  builder
    .addCase(save.pending, (state) => {
      ...
    });
},
});
...
```

Listing 15.21 Integration of the “remove” Saga into “BooksSlice” (src/features/books/booksSlice.ts)

In the final step, you adjust the dispatching of the action in the `List` component. For this purpose, you replace the previous implementation with the call of the `removeAction.request` action creator function:

```
import React, { useEffect } from 'react';
import { useSelector } from 'react-redux';
import { useNavigate } from 'react-router-dom';
import { useAppDispatch } from '../../app/hooks';
import { loadDataAction, removeAction } from './books.actions';
```

```

import {
  selectBooks,
  selectLoadingState,
  selectRemoveState,
} from './booksSlice';

const List: React.FC = () => {
  const books = useSelector(selectBooks);
  const loadingState = useSelector(selectLoadingState);
  const removeState = useSelector(selectRemoveState);
  const dispatch = useAppDispatch();
  const navigate = useNavigate();

  useEffect(() => {
    dispatch(loadDataAction.request());
  }, [dispatch]);

  if (loadingState === 'pending') {...} else {
    return (
      <>
        {removeState === 'pending' && <div>Data record will be ↩
          deleted</div>}
        {removeState === 'error' && (
          <div>An error occurred during deletion</div>
        )}
        <table>
          <thead>...          </thead>
          <tbody>
            {books.map((book) => (
              <tr key={book.id}>
                <td>{book.title}</td>
                <td>{book.author}</td>
                <td>{book.isbn}</td>
                <td>
                  <button
                    onClick={() => ↩
                      dispatch(removeAction.request(book.id))}
                  >
                    delete
                  </button>
                </td>
                <td>
                  <button onClick={() => navigate(`/edit/${book.id}`)}>
                    edit
                  </button>
                </td>
            )}
          </tbody>
        </table>
      </>
    );
  }
}

```



```

        </td>
      </tr>
    )}
  </tbody>
</table>
<button onClick={() => navigate('/new')}>new</button>
</>
);
}
};
```

`export default List;`

Listing 15.22 Dispatching the “removeAction.request” Action (src/features/books/List.tsx)

Having made these adjustments, you can load the data records from the server and use the **Delete** buttons in the list to delete the individual records. When you confirm the operation, you will see, as shown in [Figure 15.4](#) in Redux Dev Tools, that first the books/remove/pending action is triggered, and then the books/remove/fulfilled action.

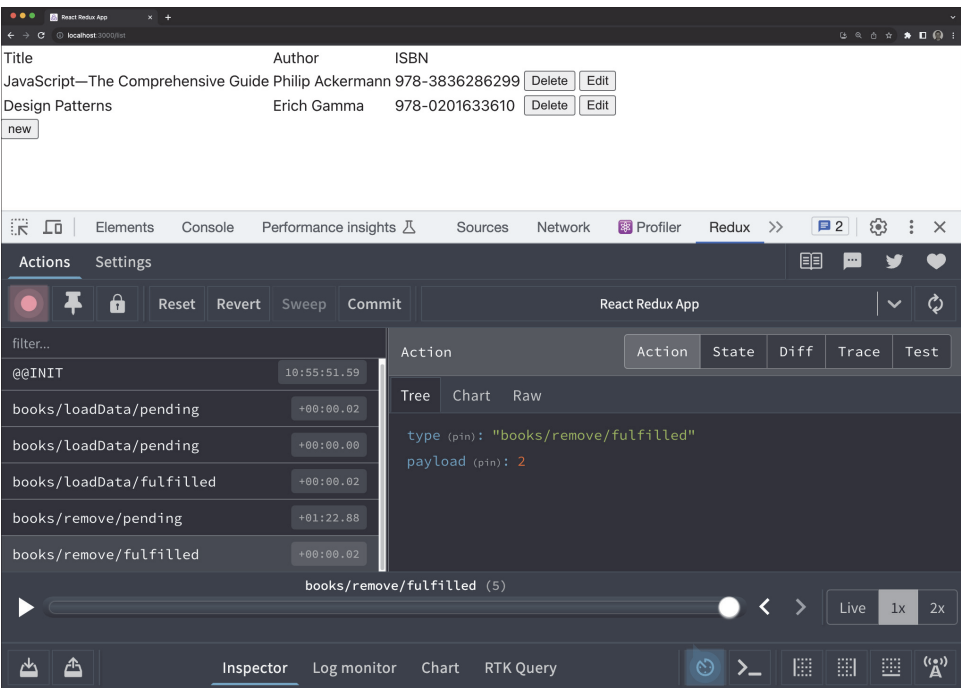


Figure 15.4 Deletion Process in Redux Dev Tools

To complete the conversion from Redux Thunk to Redux Saga, all that’s left to do is to edit and create data records.

15.3.4 Creating and Modifying Data Records Using Redux Saga

To create or edit data records, you first create a new action creator function named `saveAction` in the `books.actions.ts` file. You can see the source code of the file in [Listing 15.23](#):

```
import { createAsyncAction } from 'typesafe-actions';
import { Book, InputBook } from './Book';

export const loadDataAction = createAsyncAction(...)<void, Book[], void>();

export const removeAction = createAsyncAction(...)<number, number, void>();

export const saveAction = createAsyncAction(
  'books/save/pending',
  'books/save/fulfilled',
  'books/save/rejected'
)<InputBook, Book, void>();
```

Listing 15.23 Implementation of “saveAction” (`src/features/books/books.actions.ts`)

The `saveAction.request` action accepts an object of type `InputBook` as payload. This means that the `id` property is optional, and you can use this action for both creating and editing. The `saveAction.success` action contains a valid `Book` object as payload. For this action, you first implement a suitable saga function. In [Listing 15.24](#), you can see the implementation of the save function:

```
import { takeLatest, put } from '@redux-saga/core/effects';
import { Book } from './Book';
import { loadDataAction, removeAction, saveAction } from './books.actions';

function* loadData(): Generator {...}

function* remove({
  payload: id,
}: ReturnType<typeof removeAction.request>): Generator {...}

function* save({
  payload: book,
}: ReturnType<typeof saveAction.request>): Generator {
  try {
    let url = 'http://localhost:3001/books';
    let method = 'POST';
    if (book.id) {
      url += `/${book.id}`;
      method = 'PUT';
    }
  }
}
```

```

    }

    const response = (yield fetch(url, {
      method,
      headers: {
        'Content-Type': 'application/json',
      },
      body: JSON.stringify(book),
    })) as Response;
    if (response.ok) {
      const data = (yield response.json()) as Book;
      yield put(saveAction.success(data));
    } else {
      yield put(saveAction.failure());
    }
  } catch (e) {
    yield put(saveAction.failure());
  }
}

export default function* booksSaga() {
  yield takeLatest(loadDataAction.request, loadData);
  yield takeLatest(removeAction.request, remove);
  yield takeLatest(saveAction.request, save);
}

```

Listing 15.24 Implementation of the “save” Saga (src/features/book/books.saga.ts)

In the saga function, you check whether the data record passed in the payload of the action has an `id` property. If so, it’s an existing data record that you want to modify. In that case, you use the `PUT` method and append the value of the `id` property to the URL. For new data records, you use the `POST` method. In both cases, you use the `Fetch` API to send the data to the server and dispatch the `saveAction.success` action in case of a success or the `saveAction.failure` action in case of a failure.

In `BooksSlice`, you delete the `save-thunk` function and adjust the actions in the `extraReducers` property, as in [Listing 15.25](#):

```

import { createSelector, createSlice } from '@reduxjs/toolkit';
import { ActionType, getType } from 'typesafe-actions';
import { RootState } from '../app/store';
import { Book, InputBook } from './Book';
import { loadDataAction, removeAction, saveAction } from './books.actions';

export type BooksState = {...};

```

```

export const booksSlice = createSlice({
  name: 'books',
  initialState: {} as BooksState,
  reducers: {},
  extraReducers: (builder) => {
    builder
      .addCase(getType(loadDataAction.request), (state) => {...});

    builder
      .addCase(getType(removeAction.request), (state) => {...});

    builder
      .addCase(getType(saveAction.request), (state) => {
        state.saveState = 'pending';
      })
      .addCase(
        getType(saveAction.success),
        (state, action: ActionType<typeof saveAction.success>) => {
          if (action.payload.id) {
            const index = state.books.findIndex(
              (book) => book.id === action.payload.id
            );
            state.books[index] = action.payload as Book;
          } else {
            state.books.push(action.payload);
          }
        }
      )
      .addCase(getType(saveAction.failure), (state) => {
        state.saveState = 'error';
      });
  },
});

```

...

Listing 15.25 Integration of the Saga Actions into “BooksSlice”
(src/features/books/booksSlice.ts)

The final step of the Redux Saga integration is now to adjust the `Form` component in such a way that it dispatches the correct action—namely, the `saveAction.request` action:

```

import React, { useEffect } from 'react';
import { useSelector } from 'react-redux';
import { selectBook, selectSaveState } from './booksSlice';

```

```

import { useForm } from 'react-hook-form';
import { useNavigate, useParams } from 'react-router-dom';
import { useAppDispatch } from '../../app/hooks';
import { InputBook } from './Book';
import { saveAction } from './books.actions';

const Form: React.FC = () => {
  const getBook = useSelector(selectBook);
  const dispatch = useAppDispatch();
  const navigate = useNavigate();
  const { register, handleSubmit, reset } = useForm<InputBook>({
    defaultValues: { title: '', author: '', isbn: '' },
  });
  const saveState = useSelector(selectSaveState);

  const { id } = useParams<{ id?: string }>();

  useEffect(() => {
    if (id) {
      const book = getBook(parseInt(id, 10));
      reset(book);
    }
  }, [id, reset, getBook]);

  return (
    <>
      {saveState === 'pending' && <div>Data will be saved.</div>}
      {saveState === 'error' && <div>An error has ⬅
        occurred.</div>}

      <form
        onSubmit={handleSubmit((data) => {
          dispatch(saveAction.request(data));
          navigate('/list');
        })}
      >...</form>
    </>
  );
};

export default Form;

```

Listing 15.26 Integration of the “saveAction.request” Action into the “Form” Component (src/features/)

If you switch to the browser and load your application there, the `books/loadData/pending` action will be dispatched first, and if the server responds successfully, the `books/loadData/fulfilled` action will be dispatched. Once the list is displayed, you can create new data records or modify existing ones. In [Figure 15.5](#), you can see the Redux Dev Tools view when modifying an existing data record.

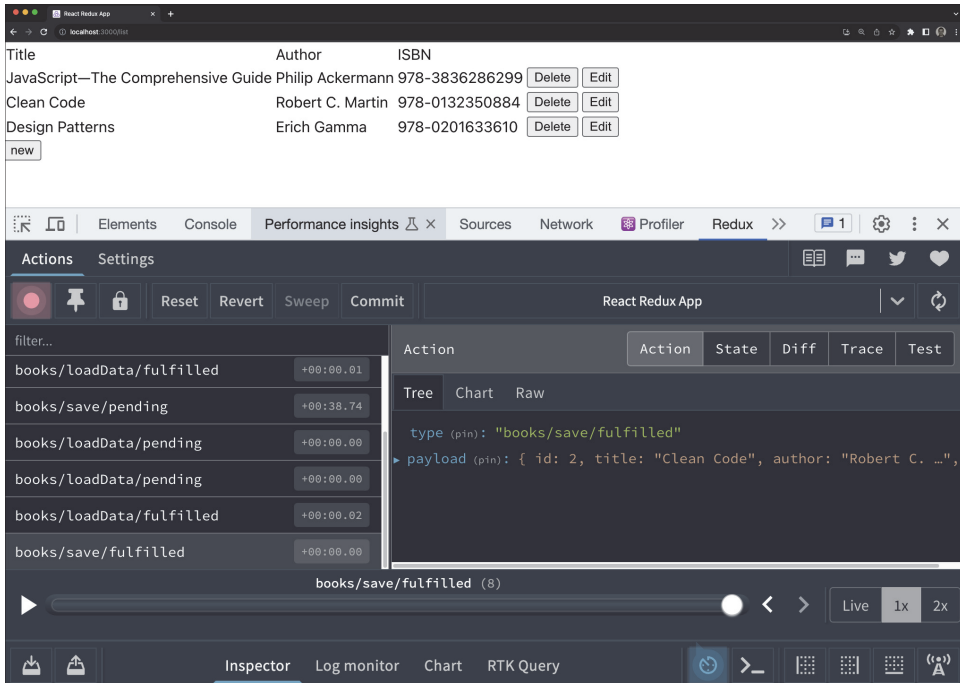


Figure 15.5 Modifying Data Records Using Redux Saga

Now that you've successfully migrated your application from Redux Thunk to Redux Saga, in the next step we'll turn our attention to the third alternative for asynchronous middleware in Redux.

15.4 State Management Using RxJS: Redux Observable

The third and probably most flexible way to deal with asynchronous operations in Redux is the *Redux Observable* library. It's based on *RxJS*, a library for reactive programming with observables.

RxJS

RxJS is an open-source library that implements Reactive Extensions for JavaScript. *Reactive Extensions* is an implementation of the Observer pattern that has been

extended with operators. The main three components of this library are observables, observers, and operators:

■ Observable

The *observable* is the data source. This can be any structure. Synchronous data sources such as arrays are possible, but so are asynchronous constructs such as promises or data streams. RxJS provides operators that turn such a structure into an observable. Alternatively, you can create such an observable using the `create` method of the `Observable` class itself. The observable can raise any number of events, which are passed to the observer via a chain of operators.

■ Operator

An *operator* is a function that can be manipulated by events on their way to their destination—the observer. There are several classes of operators. For example, there are *generating operators* that deal with the creation of observables, or *filter operators* that filter the event stream between observable and observer so that only a portion of the events arrive. Another category involves *transforming operators*, which transform values in the operator chain. RxJS allows you to combine several operators one after the other to form a chain. Each event that's triggered by the observable passes through the operator chain.

■ Observer

The *observer* is the target of the event. If it isn't removed from the data stream beforehand by a filter, then the observer receives the event—possibly modified by the operator chain—and can use it. The observer is a simple function in most cases.

For Redux Observable, the stream of actions is the observable. This way, you can register, apply operators, and finally perform certain actions.

The most important element in Redux Observable is the *epic*. This is a function that receives a stream of actions and returns a stream of actions.

15.4.1 Installing and integrating Redux Observable

Redux Observable is installed using the `npm install redux-observable rxjs` command. Separate type definitions don't need to be installed as the package is developed in TypeScript and includes appropriate type definitions. The basic setup of Redux Observable is similar to that of Redux Saga: First, you create the Epic middleware, which you then integrate into your store using the `middleware` method. In the last step, you use the `run` method of the middleware to register your epics. [Listing 15.27](#) shows these steps in the `store.ts` file:

```
import { createEpicMiddleware } from 'redux-observable';

import { configureStore } from '@reduxjs/toolkit';
import booksReducer from '../features/books/booksSlice';
```

```
import rootEpic from './rootEpic';

const epicMiddleware = createEpicMiddleware();

export const store = configureStore({
  reducer: {
    books: booksReducer,
  },
  devTools: true,
  middleware(getDefaultMiddleware) {
    return [...getDefaultMiddleware(), epicMiddleware];
  },
});

epicMiddleware.run(rootEpic);

export type AppDispatch = typeof store.dispatch;
export type RootState = ReturnType<typeof store.getState>;
```

Listing 15.27 Integration of the Epic Middleware (src/app/store.ts)

With these preparations, you can now proceed to implementing read access to the server.

15.4.2 Read Access to the Server Using Redux Observable

Redux Observable follows the same principle as Redux Saga. The epics are completely detached from all other Redux structures. For example, if you want to load data from the server, you dispatch the `loadDataAction.request` action. The Books epic responds to this by sending a corresponding request to the server. Depending on whether the feedback is positive or negative, either the `loadDataAction.success` or the `loadDataAction.failure` action will be dispatched. For you, this means that you only have to worry about the epics during implementation. The components remain unaffected by the switch from Redux Saga to Redux Observable, and you can also keep the action structures and reducers unchanged. In [Listing 15.28](#), you can see the implementation of the Books epic, which you store in the `books.epic.ts` file in the `src/features/books` directory:

```
import { combineEpics, ofType, Epic } from 'redux-observable';
import { from, of } from 'rxjs';
import { map, catchError, switchMap } from 'rxjs/operators';
import { loadDataAction } from './books.actions';

const loadData: Epic = (action$) =>
  action$.pipe(
```



```
    ofType(loadDataAction.request),
    switchMap(() =>
      from(
        fetch('http://localhost:3001/books').then((response) => {
          if (response.ok) {
            return response.json();
          } else {
            return Promise.reject();
          }
        })
      ).pipe(
        map((data) => loadDataAction.success(data)),
        catchError((err) => of(loadDataAction.failure(err)))
      )
    )
  );

export default combineEpics(loadData);
```

Listing 15.28 Implementation of the “Books” Epic to Load Data from the Server
(src/features/books/books.epic.ts)

For each operation, you implement a standalone epic function. The `loadData` function stands for read access. At the end of the file, you combine all the individual epics in this file using the `combineEpics` function and export them so that you can integrate them into the root epic.

The `loadData` function is of type `Epic`—that is, a function that receives a stream of actions and returns a stream of actions. Such data or event streams are usually identified by appending a `$` character to the variable. The `action$` parameter represents the observable. The most important method of the observable is the `pipe` method. This method enables you to create a chain of operators. First, you use the `ofType` filter operator from the `redux-observable` package to filter the action stream and process only actions of type `loadDataAction.request`.

The `switchMap` operator is a combining operator that combines an outer stream with an inner stream and returns the result. The outer stream in this case is the action stream, while the inner stream is a stream representing the result of the server call with the Fetch API. You use the `from` operator to create a new observable from the `Promise` object returned by the Fetch API. You use the `map` operator to transform the server's feedback into the `loadDataAction.success` action. If an error occurs, you catch it using the `catchError` operator and transform it into a `loadDataAction.failure` action. Due to the `switchMap` operator, the result of the outer stream is the action that originates from the inner stream.

Finally, in the root epic in the *rootEpic.ts* file within the *src/app* directory, you combine all the epics of the application and integrate them in the store as middleware. The source code of this file is shown in [Listing 15.29](#):

```
import { combineEpics } from 'redux-observable';
import booksEpic from '../features/books/books.epic';

export default combineEpics(booksEpic);
```

Listing 15.29 Source Code of the Root Epic (*src/app/rootEpic.ts*)

When you open your application with this state of the code in your browser, the data will be loaded from the server using Redux Observable. The same applies in Redux Dev Tools: the application first dispatches the action with type *books/loadData/pending* and then, once the data has become available, the *books/loadData/fulfilled* action. As with the other two asynchronous Redux middleware packages, you now implement the routine for deleting existing data records.

15.4.3 Deleting Using Redux Observable

Now that you've already made all the necessary preparations to correctly integrate Redux Observable when reading the data, the changes in the implementation of the delete feature only involve the adjustment of the Books epic. At this point, you implement another epic function similar to the *loadData* function. In the callback function of the *switchMap* operator, you have access to the action and its payload that you need to select the data record. Dispatching the subsequent action is also similar to the *loadData* function:

```
import { combineEpics, ofType, Epic } from 'redux-observable';
import { from, of } from 'rxjs';
import { map, catchError, switchMap } from 'rxjs/operators';
import { loadDataAction, removeAction } from '../books.actions';

const loadData: Epic = (action$) => ...;

const remove: Epic = (action$) =>
  action$.pipe(
    ofType(removeAction.request),
    switchMap(({ payload: id }) =>
      from(
        fetch(`http://localhost:3001/books/${id}`, {
          method: 'DELETE',
        }) .then((response) => {
          if (response.ok) {
```

```

        return Promise.resolve();
      } else {
        return Promise.reject();
      }
    })
  ).pipe(
    map(() => removeAction.success(id)),
    catchError((err) => of(removeAction.failure(err)))
  )
);

```

```
export default combineEpics(loadData, remove);
```

Listing 15.30 Deleting Data Records Using Redux Observable (src/features/books/books.epic.ts)

If you now switch to the browser, you can delete any records.

15.4.4 Creating and Editing Data Records Using Redux Observable

The final step for the full integration of Redux Observable consists of implementing the create and modify features. Again, this change takes place only in the Books epic. In the source code shown in [Listing 15.31](#), you can see the necessary adjustments to the *books.epic.ts* file:

```

import { combineEpics, ofType, Epic } from 'redux-observable';
import { from, of } from 'rxjs';
import { map, catchError, switchMap } from 'rxjs/operators';
import { loadDataAction, removeAction, saveAction } from './books.actions';

```

```
const loadData: Epic = (action$) => ...;
```

```
const remove: Epic = (action$) => ...;
```

```

const save: Epic = (action$) =>
  action$.pipe(
    ofType(saveAction.request),
    switchMap(({ payload: book }) => {
      let url = 'http://localhost:3001/books';
      let method = 'POST';
      if (book.id) {
        url += `/${book.id}`;
        method = 'PUT';
      }
    })
  );

```

```

    }

    const fetchPromise = fetch(url, {
      method,
      headers: {
        'Content-Type': 'application/json',
      },
      body: JSON.stringify(book),
    }).then((response) => {
      if (response.ok) {
        return response.json();
      } else {
        return Promise.reject();
      }
    });

    return from(fetchPromise).pipe(
      map((data) => saveAction.success(data)),
      catchError((err) => of(saveAction.failure(err)))
    );
  });
}

export default combineEpics(loadData, remove, save);

```

Listing 15.31 Creating and Modifying Data Records Using Redux Observable

In the `save` epic, you use the `ofType` operator to ensure that you respond only to `saveAction.request` actions. Within the `switchMap` operator, you distinguish between new and existing data records and send the appropriate request to the server. Once the response from the server has been received, you check if the request was successful. If that's not the case, you use the `Promise.reject` method to signal an error. If successful, the `Promise` object will be resolved with the new or modified data. You convert the `Promise` object from the server communication into an RxJS stream using the `from` operator and then dispatch the `saveAction.success` action using the `map` operator in case of success or the `saveAction.failure` action using the `catchError` operator in case of a failure.

If you now switch to the browser, you can use the full functionality of the book management with Redux Observable. The image in Redux Dev Tools is not different from the one in Redux Saga. For each asynchronous action, an introductory action is always dispatched, followed by the success action. [Figure 15.6](#) shows the Redux Dev Tools view after you've created and then deleted a data record.

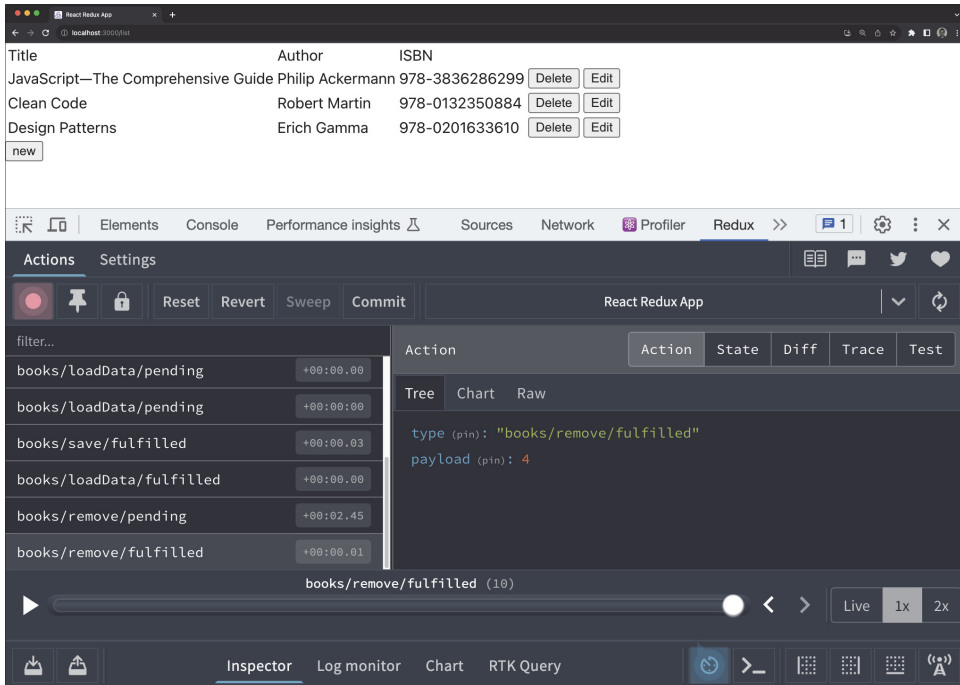


Figure 15.6 Creating and Deleting Data in Redux Dev Tools

In the following section, you'll learn how to protect your application from unauthorized access with a simple authentication process.

15.5 JSON Web Token for Authentication

The following example shows how you can protect the backend interface of your application using *JSON Web Token* (JWT). As the name suggests, JWT is based on the JSON format. The token is a standardized format that can be used, for example, to log in users. The token is divided into three parts:

- The type and the signature algorithm are specified in the *header*.
- The *payload section* can store user data to which both the client and the server have access.
- The *signature* can be used to determine whether the token is valid and has not been tampered with in any way.

In the running example, we use an unlimited valid token. In practice, you should make sure that the token has the shortest possible runtime, because if a potential attacker gets hold of the token, they'll have unlimited access. The token can't be invalidated. If a token is assigned an expiration time, the client must renew the access token after a

certain period of time. A refresh token is available for this purpose. In the renewal process, the server has the option to refuse the issuing of the access token.

To log into the backend and receive the token, you need a `Login` component through which you can send the credentials. You store the access token and any login errors in the Redux store. For the server communication, you use `Redux Observable`. The following steps are in a way a repetition of the topic of `Redux`: you implement your own slice including asynchronous server communication.

Before you start implementing the `Login` component, you need to create the required Redux structures, starting with the actions that you implement in the `login.actions.ts` file in the `src/features/login` directory:

```
import { createAction } from 'typesafe-actions';

export const loginAction = createAction(
  'login/doLogin/pending',
  'login/doLogin/fulfilled',
  'login/doLogin/rejected'
)<{ username: string; password: string }, string, void>();
```

Listing 15.32 Implementation of the Login Action (`src/features/login/login.actions.ts`)

All you need is an asynchronous action to perform the login. Initially, you pass the user name and password and then you get back either the token or an error. The server communication is taken care of by the `Login` epic that you create in the `login.epic.ts` file in the `src/features/books` directory.

```
import { combineEpics, ofType, Epic } from 'redux-observable';
import { from, of } from 'rxjs';
import { map, catchError, switchMap } from 'rxjs/operators';
import { loginAction } from './login.actions';

const login: Epic = (action$) =>
  action$.pipe(
    ofType(loginAction.request),
    switchMap(({ payload: credentials }) =>
      from(
        fetch('http://localhost:3001/login', {
          method: 'POST',
          headers: {
            'Content-Type': 'application/json',
          },
          body: JSON.stringify(credentials),
        }).then((response) => {
          if (response.ok) {
```

```
        return response.text();
      } else {
        return Promise.reject();
      }
    })
  ).pipe(
    map((data) => loginAction.success(data)),
    catchError((err) => of(loginAction.failure(err)))
  )
);
```

```
export default combineEpics(login);
```

Listing 15.33 Login Epic (src/features/login/login.epic.ts)

You then integrate the epic into `RootEpic` so that it is correctly integrated into the application. The code required for this is shown in [Listing 15.34](#):

```
import { combineEpics } from 'redux-observable';
import booksEpic from '../features/books/books.epic';
import loginEpic from '../features/login/login.epic';
```

```
export default combineEpics(booksEpic, loginEpic);
```

Listing 15.34 Integration of “loginEpic” into “RootEpic” (src/app/rootEpic.ts)

To make sure that the token you receive from the server is available in the application, you must store it in the Redux Store. The only way to do this is to use a reducer, which you implement in a separate `LoginSlice`. In [Listing 15.35](#), you can see the source code of the `loginSlice.ts` file from the `src/features/login` directory:

```
import { createSlice } from '@reduxjs/toolkit';
import { ActionType, getType } from 'typesafe-actions';
import { RootState } from '../../app/store';
import { loginAction } from './login.actions';
```

```
type LoginState = {
  token: string;
  loginState: null | 'pending' | 'completed' | 'error';
};
```

```
export const loginSlice = createSlice({
  name: 'login',
  initialState: {
    token: '',
```

```

    loginState: null,
  } as LoginState,
  reducers: {},
  extraReducers: (builder) => {
    builder
      .addCase(getType(loginAction.request), (state) => {
        state.loadingState = 'pending';
      })
      .addCase(
        getType(loginAction.success),
        (state, action: ActionType<typeof loginAction.success>) => {
          state.loadingState = 'completed';
          state.token = action.payload;
        }
      )
      .addCase(getType(loginAction.failure), (state) => {
        state.loadingState = 'error';
      });
  },
});

export const selectToken = (state: RootState) => state.login.token;
export const selectLoginState = (state: RootState) =>
  state.login.loginState;

export default loginSlice.reducer;

```

Listing 15.35 Implementation of “loginSlice” (src/features/login/loginSlice.ts)

You need to integrate `loginSlice` also into your application. This is done in the `store.ts` file in the `src/app` directory where you configure the store. The changes to this file are shown in [Listing 15.36](#):

```

import { createEpicMiddleware } from 'redux-observable';

import { configureStore } from '@reduxjs/toolkit';
import booksReducer from '../features/books/booksSlice';
import loginReducer from '../features/login/loginSlice';
import rootEpic from './rootEpic';

const epicMiddleware = createEpicMiddleware();

export const store = configureStore({
  reducer: {
    books: booksReducer,

```



```
    login: loginReducer,
  },
  devTools: true,
  middleware(getDefaultMiddleware) {
    return [...getDefaultMiddleware(), epicMiddleware];
  },
});

epicMiddleware.run(rootEpic);

export type AppDispatch = typeof store.dispatch;
export type RootState = ReturnType<typeof store.getState>;
```

Listing 15.36 Extension of the store.ts File (src/app/store.ts)

With these extensions, you've laid the groundwork for logging into your backend. But your users won't see any of that yet. So in the next step, you need to implement the Login component and connect it to Redux:

```
import React, { ChangeEvent, FormEvent, useState } from 'react';
import { useAppDispatch } from '../../app/hooks';
import { loginAction } from './login.actions';

const Login: React.FC = () => {
  const dispatch = useAppDispatch();

  const [credentials, setCredentials] = useState({
    username: '',
    password: '',
  });

  function handleChange(event: ChangeEvent<HTMLInputElement>): void {
    setCredentials((prevCredentials) => ({
      ...prevCredentials,
      [event.target.name]: event.target.value,
    }));
  }

  function handleSubmit(event: FormEvent<HTMLFormElement>) {
    event.preventDefault();
    dispatch(loginAction.request(credentials));
  }

  return (
    <form onSubmit={handleSubmit}>
```

```

    <div>
      <label htmlFor="username">Username: </label>
      <input
        type="text"
        value={credentials.username}
        onChange={handleChange}
        name="username"
        id="username"
      />
    </div>
    <div>
      <label htmlFor="password">Password: </label>
      <input
        type="password"
        value={credentials.password}
        onChange={handleChange}
        name="password"
        id="password"
      />
    </div>
    <button type="submit">submit</button>
  </form>
);
};

export default Login;

```

Listing 15.37 Implementation of the “Login” Component (src/features/login/Login.tsx)

The form from [Listing 15.37](#) is a simple React form with controlled components. Submitting the form causes the `loginAction.request` to be dispatched with the login data and processed by `LoginEpic`. The epic then makes sure that the credentials are sent to the `/login` endpoint on the server via HTTP POST. The server responds with a JWT, which you then pass to the reducer via the `loginAction.success` action, which stores it in the store. This structure allows you to integrate the `Login` component into the `App` component of the application and display either the login screen or the desired component, such as the list, depending on the presence of the token. You can see the necessary adjustments to the `App` component in [Listing 15.38](#):

```

import React from 'react';
import './App.css';
import List from './features/books/List';
import Form from './features/books/Form';
import { BrowserRouter, Routes, Route, Navigate } from 'react-router-dom';
import { useSelector } from 'react-redux';

```

```
import { selectToken } from './features/login/loginSlice';
import Login from './features/login/Login';

const App: React.FC = () => {
  const token = useSelector(selectToken);

  return (
    <BrowserRouter>
      <Routes>
        <Route path="/edit/:id" element={
          {token ? <Form /> : <Login />} />
        <Route path="/new" element={token ? <Form /> : <Login />} />
        <Route path="/list" element={token ? <List /> : <Login />} />
        <Route path="/" element={
          {
            token ? <Navigate to="/list" /> : <Login />
          } />
        </Routes>
      </BrowserRouter>
    );
  };
};

export default App;
```

Listing 15.38 Integration of the Authentication Feature in the “App” Component (src/App.tsx)

You use the `selectToken` selector to read the token from the Redux store. If it’s available, you render the respective requested component behind the different routes. But if the token hasn’t been loaded yet, you display the `Login` component so that users can obtain a token.

At the moment, the application doesn’t work because the backend—that is, `json-server`—doesn’t support authentication. However, you can fix this issue by extending the backend. For this purpose, you create a new directory inside your application named *backend* and move the *data.json* file there. Then you create a new *package.json* file in the *backend* directory using the `npm init -y` command, in which you insert the type key with the module value. Then, also in the *backend* directory, you install a number of additional packages using the `npm install cors express json-server jsonwebtoken` command and then implement your server in the *index.js* file in the *backend* directory. The source code of this file is shown in [Listing 15.39](#):

```
import express from 'express';
import jsonServer from 'json-server';
import jwt from 'jsonwebtoken';
import cors from 'cors';
```

```

const port = 3001;
const secret = 'topSecret';

const server = express();

server.use(express.json());
server.use(cors());

server.post('/login', (req, res, next) => {
  if (req.body.username === 'admin' && req.body.password === 'test') {
    res.send(
      jwt.sign({ user: req.body.username }, secret, { expiresIn: '1800s' })
    );
  }
});

server.use('/', (req, res, next) => {
  try {
    const token = req.headers['authorization'].split(' ')[1];
    jwt.verify(token, secret);
    next();
  } catch (e) {
    res.sendStatus(403);
  }
});

server.use('/', jsonServer.router('data.json'));

server.listen(port, () =>
  console.log(`server is listening to http://localhost:${port}`)
);

```

Listing 15.39 Implementation of the Backend (backend/index.js)

If you start the backend in the *backend* directory using the `node index.js` command, you can log in via the frontend and get a token in return. You'll also be correctly redirected to the list view, but you'll receive an error message there because you're requesting the data without a valid token. This problem can be fixed by extending *booksEpic*, reading the token from the store, and integrating it into the various requests. You can see how to achieve this in the *books.epic.ts* file in [Listing 15.40](#):

```

import { combineEpics, ofType, Epic } from 'redux-observable';
import { from, of } from 'rxjs';
import { map, catchError, switchMap } from 'rxjs/operators';
import { selectToken } from '../login/loginSlice';

```

```
import { loadDataAction, removeAction, saveAction } from './books.actions';

const loadData: Epic = (action$, state$) =>
  action$.pipe(
    ofType(loadDataAction.request),
    switchMap(() =>
      from(
        fetch('http://localhost:3001/books', {
          headers: {
            Authorization: `Bearer ${selectToken(state$.value)}`,
          },
        }).then((response) => {...})
      ).pipe(...)
    )
  );

const remove: Epic = (action$, state$) =>
  action$.pipe(
    ofType(removeAction.request),
    switchMap(({ payload: id }) =>
      from(
        fetch(`http://localhost:3001/books/${id}`, {
          method: 'DELETE',
          headers: {
            Authorization: `Bearer ${selectToken(state$.value)}`,
          },
        }).then((response) => {...})
      ).pipe(...)
    )
  );

const save: Epic = (action$, state$) =>
  action$.pipe(
    ofType(saveAction.request),
    switchMap(({ payload: book }) => {
      ...
      const fetchPromise = fetch(url, {
        method,
        headers: {
          'Content-Type': 'application/json',
          Authorization: `Bearer ${selectToken(state$.value)}`,
        },
        body: JSON.stringify(book),
      }).then((response) => {...});
    });
  );
```

```

    return from(fetchPromise).pipe(...);
  })
);

export default combineEpics(loadData, remove, save);

```

Listing 15.40 Integration of the Token into the “Books” Epic
(src/features/books/books.epic.ts)

15.6 Summary

In this chapter, you modified your previously local Redux application to communicate with the server. In the process, you learned about different variants of Redux middleware:

- *Middleware* packages are extensions for Redux that allow you to handle asynchronous operations, for example.
- *Redux Thunk*, *Redux Saga*, and *Redux Observable* are the three most popular async middleware implementations for Redux.
- Redux Thunk employs *action creators*, which return functions instead of simple objects, and it’s a direct part of the Redux Toolkit.
- Redux Saga is based on *ECMAScript generators*, which are used to process asynchronous operations.
- Redux Observable uses RxJS to map asynchronous operations over asynchronous data streams.
- Redux Saga and Redux Observable work with similar principles: *sagas* and *epics* respond to actions, perform the respective operations, and then dispose of their own actions, to which the reducer then responds.
- The middleware functions for Redux have very few to no contact points with the components of the application. The only connection is the actions, which are dispatched by the components, and the state, which is changed by the middleware again via actions.

Until now, you’ve always communicated with the server using ordinary HTTP, addressing RESTful APIs. *GraphQL* is an alternative to this communication pattern, and in the next chapter we’ll show you how to integrate GraphQL into your application.

Chapter 16

Server Communication Using GraphQL and the Apollo Client

GraphQL is a query language that is increasingly being used on the web. The language is used for communication between client and server and is an alternative to REST and SOAP. GraphQL was developed by Facebook and released in 2015, but GraphQL is available under the MIT license and part of the Linux Foundation. GraphQL enables you to read data from the server and also create, manipulate, and delete data records.

Throughout this chapter, you'll learn what GraphQL can do for you and how you can integrate this query language into your React application.

16.1 Introduction to GraphQL

GraphQL is still far from REST's numbers in terms of its prevalence in web development. With GitHub, one major company already offers a public GraphQL interface—and for good reason, because the language provides a high degree of flexibility. When using GraphQL, you have to make significant changes compared to REST APIs because GraphQL has a very different approach to standard web issues.

16.1.1 The Characteristics of GraphQL

To get the most out of using GraphQL, you should keep the following in mind:

- **With your request, you determine the structure of the response.**
With a REST resource, the structure is predetermined by the server for each request, even if you only need parts of it. With GraphQL, you specify the exact fields you need in the request. This enables you to reduce the transfer volume.
- **GraphQL uses graphs.**
The individual objects stored by a web server rarely stand alone, but have connections to each other. If you consider a REST interface, you must load the original object and its connected objects in separate queries. Using GraphQL, you can resolve these links in a single query. Thus, entire object trees can be loaded in one query using this graph mapping.

- **GraphQL contains a type system.**

GraphQL supports a number of scalar data types, such as integers, floats, strings, and Booleans, as well as objects and lists of types equivalent to a JavaScript array.

- **GraphQL is only a query language.**

GraphQL is just an interface, just as REST is also just an interface. The business logic of your web server as well as the data management remain unaffected by GraphQL.

- **The schema of GraphQL can be used as documentation.**

The schema of GraphQL serves a similar purpose as the OpenAPI specification for REST interfaces. It can be queried and provides hints about the linkage of the individual objects and the types of the properties.

- **GraphQL eliminates the need for interface versioning.**

You can consider the schema of a GraphQL interface as living documentation. Fields can always be added during the lifecycle of the interfaces. If a client doesn't query these, there won't be any disadvantages as a result. Fields that are no longer needed are marked as *deprecated* and can be removed from the interface after a certain period of time.

But despite all the advantages and improvements that come with GraphQL, it also has a number of disadvantages.

16.1.2 The Disadvantages of GraphQL

To make the best use of a technology, library, or even a design pattern, it doesn't suffice to just know its advantages. You also need to know about its disadvantages and limitations so that you can properly make the decision to use it and take measures to compensate for any disadvantages when doing so. GraphQL does have some drawbacks, some of which are due to the design of the query language:

- **Overhead**

Each abstraction layer in an application implies some overhead. In the case of GraphQL, you add an additional layer on both the client and server side. On the client side, you formulate a query that must then be interpreted and resolved on the server side.

- **No caching**

GraphQL uses HTTP as a transport protocol. However, the caching mechanisms provided by the protocol are not effective here as the requests are not differentiated by the URI, but solely by the query that is formulated in the request. This disadvantage can be eliminated by caching within the server implementation and also plays less of a role in a single-page application as permissions also need to be checked frequently here.

- **Plain text**

The schema of GraphQL supports only a handful of simple data types. More advanced operations such as file uploads are not provided for in the standard GraphQL version. However, various libraries such as Apollo also provide solutions for this.

If you approach the topic of GraphQL, you'll quickly come across a number of technical terms, like *query*, *mutation*, and *schema*. Before we get into the actual implementation, you'll learn the meaning behind these terms.

16.1.3 The Principles of GraphQL

GraphQL requires certain structures so that the interface can be built and made available. The greater the flexibility in the client, the more effort must be expended on the server side.

Defining the Data Structure Using the GraphQL Schema

You can use a *schema* to define the structure of your GraphQL interface. The schema defines the individual objects and properties as well as their types. All information listed in the schema can be queried by the client. However, this also means that the information that is not included in the schema cannot be read. This is where the flexibility of GraphQL reaches its limit. The interface cannot be extended dynamically at runtime.

In [Listing 16.1](#), you can see a basic version of the GraphQL schema for the user interface:

```
type Book {
  id: ID!
  title: String!
  isbn: String
  author: Author
}

type Author {
  id: ID!
  firstname: String
  lastname: String
}

type Query {
  book(title: String isbn: String): [Book];
}
```

Listing 16.1 User Schema

The schema consists of three types: **Book**, **Author**, and **Query**. The **Book** type represents the individual book records that can be queried through the system. The **Author** type defines the structure of an author of a book. The **Query** type is a GraphQL-specific type; you'll learn more about this topic in the "[Read Data with Queries](#)" section.

To define the schema, you access the data types of GraphQL. These are similar to the types provided by TypeScript. The only peculiarities in the schema that need further explanation are the ID type, which stands for a unique `string` value; the exclamation mark after some types, which means that this type must not take the value `null`; and the `Book` type inside the `Query` type, which is enclosed by square brackets. This type specification means that this is a list; the data type is comparable to an array in JavaScript.

Resolver: The Link between GraphQL and the Application

In the broadest sense, the GraphQL interface works like a facade in front of the application. The schema defines the structure of the data that can be used via the interface. The link between GraphQL and the rest of the application is formed by the *resolvers*. These are functions that ensure that the desired information is read or written. Resolvers can be parameterized to provide additional flexibility. You have already seen this parameterization in the schema from [Listing 16.1](#) in the `Query` type with the `book` key. There, the possible parameters as well as the associated types are defined in parentheses. You have access to them within the resolver. In [Listing 16.2](#), you can see an example of such a resolver function:

```
function getBook(title = '', isbn = '') {  
  let books = db.get('books');  
  
  if (title) {  
    books = books.filter({ title });  
  }  
  
  if (isbn) {  
    books = books.filter({ isbn });  
  }  
  
  const result = books.value();  
  
  return result.map(book => {  
    return { ...book, author: getAuthor(book.author_id) };  
  });  
}
```

Listing 16.2 Resolver Function

The source code comes from the GraphQL backend for the Books interface. This application is based on the `express`, `graphql`, and `express-graphql` packages. Concerning saving, this application uses the `lowdb` package to store the information in a JSON file. The complete source code of the backend can be found in the **Downloads** section on this book's webpage.

Read Data with Queries

GraphQL distinguishes between read and write access. Read access is mapped via the query type. In such a query, you define the structure of the data you expect to be returned and also have the option of specifying filter criteria to further restrict the result area. For the formulation of such queries GraphQL uses its own format. By mapping the dependencies between individual objects in the schema, you can also query child objects in a query. The prerequisite for this is that the schema supports this. [Listing 16.3](#) shows a concrete example of a query:

```
query {
  book(title:"Clean Code") {
    title
    isbn
    author{
      lastname
    }
  }
}
```

Listing 16.3 Read Query in GraphQL

The query starts with the `query` keyword. This is followed by an object structure with the actual request. In this case, a list of books is to be read that have the title *Clean Code*. Of the data records returned, the title, ISBN, and author's last name are required. No other user information is delivered by the server.

For development using GraphQL, there is a browser-based tool, *GraphiQL*, which you can use to test your queries. The backend delivers the GraphiQL interface via the following address: <http://localhost:3001/api>.

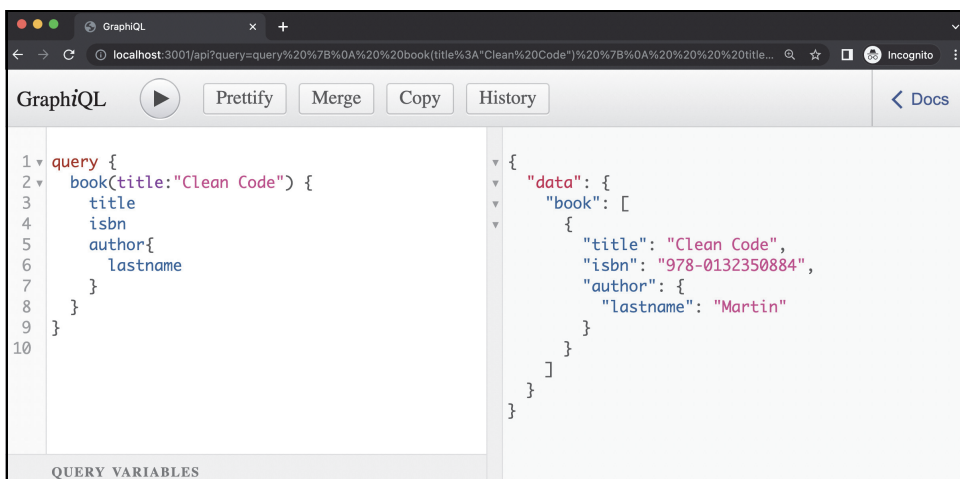


Figure 16.1 GraphiQL User Interface

Figure 16.1 shows the GraphiQL user interface at the moment when the query from Listing 16.3 is launched. In the left-hand pane of the window, you can formulate your requests. The tool supports you with features like autocompletion and error highlighting. The right-hand pane is used to display the results. GraphiQL is a development tool that you can access in the environment using the `express-graphql` package. However, you need to enable it in the configuration. For the production use of your interface, you should make sure that GraphiQL isn't available to every user, as access is a potential security risk. You can either block the external access to the interface or completely disable the tool for the production application.

Write Access via Mutations

Creating new data records and editing and deleting existing records is covered by the `Mutation` type. In the broadest sense, a mutation request works like a function. You pass certain parameters to it, which are validated against the schema and then passed to the resolver function. This function then takes care of passing them on to the actual application logic. Listing 16.4 contains an example of a mutation type that you can use to create, edit, and also delete a data record:

```
type Mutation {  
  deleteBook(id: String!): Boolean;  
  createBook(book: InputBook!): Book;  
  updateBook(updatedBook: Book!): Book;  
}
```

Listing 16.4 Examples of Mutations

When generating the book, you'll see another special feature of the GraphQL interface: the input type is not the previously used `Book` type, but a separate `InputBook` type. This type is referred to as an *input type*. It largely corresponds to the structure of the `Book` type, but it doesn't contain all mandatory fields such as the ID, as this info isn't yet available at the time of creation.

In the sample code for the book, you'll find the executable GraphQL backend. To start it, go to the *backend* directory and run the `npm install` command to download and install all the required packages. After that, you can start the backend using the `npm start` command. The backend can then be reached via the following address: `http://localhost:3001/api`. With this preparation, the next step is to get started with the connection in the frontend.

16.2 Apollo: A GraphQL Client for React

Basically, you don't need an additional library to communicate with a GraphQL server. You can also formulate a request using the Fetch API of the browser. The disadvantage of this method is that you have to take care of all the tasks, such as composing the

request, yourself. Numerous clients for GraphQL have emerged in the JavaScript community in the past. Two of the most popular solutions are Relay and the Apollo client. *Relay* is a GraphQL client developed by Facebook. *Apollo* is an independent, community-driven project. In this chapter, we'll focus on using the Apollo client, which is currently more widely used than Relay.

Apollo is a project that provides both server-side and client-side solutions for GraphQL. However, the Apollo client can be used independently of the server and is capable of working with any GraphQL server. The Apollo client is a central connection to the GraphQL server in the application, through which the queries and mutations are then sent. Like most other React libraries, the Apollo client provides you with a Hooks API that you can use to communicate with the GraphQL API from within your components. In this chapter, we'll implement read and write access to the GraphQL server with this interface.

16.2.1 Installation and Integration into the Application

You install the Apollo client via your package manager, just like the other extensions for React. The following examples are based on a new React app that you initialized via the `npx create-react-app` library `--template typescript` command. After creating the application, you use the following command to install the required packages: `npm install @apollo/client graphql`. The individual packages perform the following tasks:

- **@apollo/client**

This package contains everything you need to communicate with a GraphQL server from within your React application. It contains, for example, the necessary hooks, but also other features, such as a memory cache or the interfaces for local state management as you know it from Redux.

- **graphql**

The `graphql` package is used for parsing GraphQL queries. This is the reference implementation in JavaScript. This package can be used both on the client side and on the server side.

The Apollo client uses the Context API of React to configure the connection to the server in a central location and then make it available across the entire application. For this purpose, you usually include the client in the root component of the app. [Listing 16.5](#) shows the Apollo client configuration in the App component:

```
import React from 'react';
import './App.css';
import List from './List';

import { ApolloClient, InMemoryCache, ApolloProvider } from
  '@apollo/client';
```

```
const client = new ApolloClient({
  uri: 'https://localhost:3001/api',
  cache: new InMemoryCache(),
});

const App: React.FC = () => {
  return (
    <ApolloProvider client={client}>
      <List />
    </ApolloProvider>
  );
};

export default App;
```

Listing 16.5 Integration of the Apollo Client in the Application (src/App.tsx)

You use the `ApolloClient` class from the `@apollo/client` package to create a new instance of the Apollo client. The minimal configuration you need to specify for this example consists of an object where you specify the address of the GraphQL interface via the `uri` property, and the `cache` property with an instance of the `InMemoryCache` class, which Apollo uses to cache the results. You pass the client instance to the `ApolloProvider` component via the `client` prop. At this point, the Apollo client is configured and ready for use. The `List` component currently consists of a placeholder, which you'll replace with dynamic content in the next step.

16.2.2 Read Access to the GraphQL Server

The first feature you implement with the Apollo client is a list view of all `Book` records in the application. For this purpose, you formulate a query to the server and load all records. However, unlike a traditional request in a REST interface, in this case you read only a portion of the `Book` properties. In [Listing 16.6](#), you can see the source code of the `List` component that's responsible for displaying the list:

```
import { gql, useQuery } from '@apollo/client';
import React from 'react';

type Book = {
  id: string;
  title: string;
  isbn: string;
};

const booksQuery = gql`
```

```

    query {
      book {
        id
        title
        isbn
      }
    }
  `;

const List: React.FC = () => {
  const { data } = useQuery<{ book: Book[] }>(booksQuery);

  return (
    <table>
      <thead>
        <tr>
          <th>Title</th>
          <th>ISBN</th>
        </tr>
      </thead>
      <tbody>
        {data?.book.map((book) => (
          <tr key={book.id}>
            <td>{book.title}</td>
            <td>{book.isbn}</td>
          </tr>
        ))}
      </tbody>
    </table>
  );
};

export default List;

```

Listing 16.6 List Representation of Data from a GraphQL Interface (src/List.tsx)

Due to `ApolloProvider` from [Listing 16.5](#), the connection to the GraphQL server is available across the entire application, and you can use the query hook in your components. You pass a GraphQL query to this hook, which you can create using the `gql` function. This function is a tagging function for a template string. This JavaScript feature ensures that the passed template string is processed by the specified function and a new string is returned.

The query itself is a standard query that you can also submit in this form in `GraphiQL`, which can help you both in formulating a more extensive query and in debugging. As a

result of the query hook, you get an object with various properties. You can access the data of the interface via the `data` key. The data becomes available as soon as the server's response is available. The hook makes sure that the component is then re-rendered with the data. In the query, the `id`, `title`, and `isbn` data of a book is read. All other information that would be available for a book through the interface, such as author data, isn't transmitted by the server because it wasn't requested by the client. [Figure 16.2](#) shows the rendered book list.

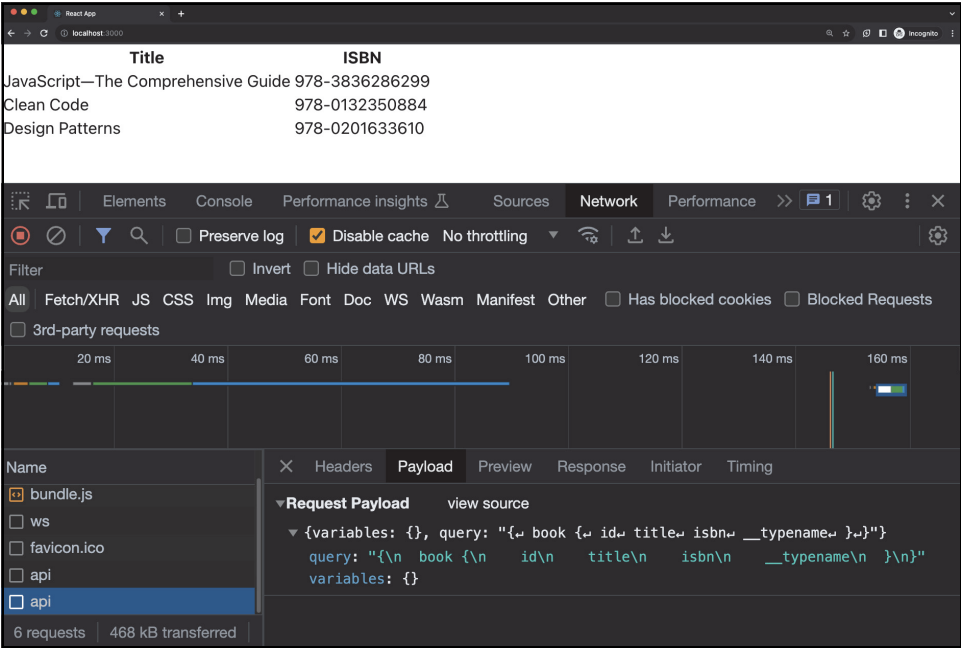


Figure 16.2 List View of the Book Records

In addition to the requested data, the query hook provides additional information that can be used to increase the usability of your application.

16.2.3 States of a Request

The object returned by the query hook has some more information about the state of the query. The `loading` property indicates whether the request is currently being processed and the data is being loaded. In this case, the property has the value `true`. Once the data is available, the value is set to `false` by the library. Typically, you use this information to display a loading indicator that signals a short wait time to the user. In the best case, the data is available so quickly that the user doesn't consciously notice the loading indicator.

The second property that's relevant for you besides `data` and `loading` is the `error` property. It contains an object of the `ApolloError` type if an error occurred during the query.

In production operation, you should be careful not to give too much information about your application to the outside world, and instead of a concrete technical error message, it's better to integrate a general message with the option to contact you. In [Listing 16.7](#), both properties are integrated in the `List` component:

```
import { gql, useQuery } from '@apollo/client';
import React from 'react';

type Book = { ... };

const booksQuery = gql`...`;

const List: React.FC = () => {
  const { data, loading, error } = useQuery<{ book: Book[] }>(booksQuery);

  if (loading) {
    return <div>Loading...</div>;
  }

  if (error) {
    return <div>An error has occurred.</div>;
  }

  return (
    <table>
      <thead>
        <tr>
          <th>Title</th>
          <th>ISBN</th>
        </tr>
      </thead>
      <tbody>
        {data?.book.map((book) => (
          <tr key={book.id}>
            <td>{book.title}</td>
            <td>{book.isbn}</td>
          </tr>
        ))}
      </tbody>
    </table>
  );
};

export default List;
```

Listing 16.7 Display of a Loading Indicator and an Error Message (src/List/List.tsx)

While loading the data, you display the `Loading...` text. If an error occurs—that is, if an error object is present—you also display a corresponding message. Only if both the `charge` state is `false` and there is no error will you display the list.

As you can already see in the example, the query hook is implemented as a generic function to which you can pass a type for the server response. However, Apollo supports you much further when using TypeScript.

16.2.4 Type Support in the Apollo Client

The hooks from the `@apollo/client` package are generic functions. This means that you can define types for these functions. For example, the query hook accepts two types: the first type represents the data structure that is returned, and the optional second type represents variables that are used in the query. But because each GraphQL query potentially produces different data structures, this means that you must also define a new type for each query.

If your entire application is based on GraphQL, the generation and management of types can become very time-consuming and also error-prone. For this reason, you can draw on tools that will do this task for you. The tool that solves this problem for you is called *GraphQL Code Generator*. This command-line tool allows you to have a type-safe version of hooks generated for the various requests. To use GraphQL Code Generator, you need to install some packages via the following command: `npm install @graphql-codegen/typescript-react-apollo @graphql-codegen/typescript @graphql-codegen/typescript-operations`.

The next step consists of creating a file named `codegen.yml` in the root directory of your application. The contents of this file are shown in [Listing 16.8](#):

```
schema: http://localhost:3001/api
documents: './src/**/*.tsx'
generates:
  src/graphql/generated.ts:
    plugins:
      - typescript
      - typescript-operations
      - typescript-react-apollo
    config:
      withHooks: true
```

Listing 16.8 Configuration of GraphQL Code Generator (`codegen.yml`)

Based on this configuration file, GraphQL Code Generator searches for GraphQL queries in all TSX files and generates hook functions and their associated types for you. These hooks are located in the `src/graphql/generated.ts` file. If possible, you shouldn't modify this file because the command-line tool overwrites it every time it's called. For

a more comfortable working setup with GraphQL Code Generator, you can create an entry in the *package.json* file of your application with the *generate* key and the *graphql-codegen* value. After that you can generate the hooks using the `npm run generate` command.

In the current version of the query, the generation of the hooks doesn't yet work ideally as the query is still anonymous; that is, it doesn't have a name. However, you can fix this problem in [Listing 16.9](#) by giving it the name `BooksList` and including the generated hook function right away for the list view:

```
import { gql } from '@apollo/client';
import React from 'react';
import { useBooksListQuery } from './graphql/generated';

const booksQuery = gql`
  query BooksList {
    book {
      id
      title
      isbn
    }
  }
`;

const List: React.FC = () => {
  const { data, loading, error } = useBooksListQuery();
  if (loading) {
    return <div>Loading...</div>;
  }

  if (error) {
    return <div>An error has occurred.</div>;
  }

  return (
    <table>
      <thead>
        <tr>
          <th>Title</th>
          <th>ISBN</th>
        </tr>
      </thead>
      <tbody>
        {data?.book?.map(
          (book) =>
```

```
      book && (  
        <tr key={book.id}>  
          <td>{book.title}</td>  
          <td>{book.isbn}</td>  
        </tr>  
      )  
    )}  
  </tbody>  
</table>  
);  
};
```

```
export default List;
```

Listing 16.9 Integration of the Generated Hook Function (src/List.tsx)

If you run the `npm run generate` command again, GraphQL Code Generator will generate the *generated.ts* file for you in the *src/graphql* directory. Among other things, this file contains the `useBooksListQuery` function. The name derives from the name of the query—in this case, `BooksList`—and from the `use` prefix to indicate that the function is a hook function, and from the postfix query, which stands for a GraphQL query.

As the types strictly follow the specifications in the schema, you have to integrate further checks in the list representation because, for example, a single record can be `null` or `undefined`, which would cause an error. At this point, you’ve completed the read access to your GraphQL interface and can now turn to the write accesses.

16.2.5 Deleting Data Records

For write access, you use the mutation hook from the `@apollo/client` package. It works similarly to the query hook. The biggest difference is that instead of a data object, you get a function as a return value that you can execute. The `deleteUser` mutation of the GraphQL server is parameterized. This means that you must pass the ID of the record to be deleted as a variable to the mutation. Once the data record has been deleted, the display is supposed to be updated. You can do this by passing a configuration object to the mutation hook and specifying the list query via the `refetchQueries` option.

In preparation for deleting the records, you swap out the two queries, both the list query and the delete mutation, to a new file named *list.graphql.ts*. The source code of this file is shown in [Listing 16.10](#):

```
import gql from 'graphql-tag';  
  
gql`  
  query BooksList {
```

```

    book {
      id
      title
      isbn
    }
  }
`

graphql`
  mutation deleteBook($id: ID!) {
    deleteBook(id: $id) {
      id
    }
  }
`

```

Listing 16.10 GraphQL Queries for the List (src/list.graphql.ts)

Because you let GraphQL Code Generator generate the hook functions again, you don't need to export the queries. It's enough to formulate them locally in the file.

In the `deleteBook` mutation, you can use the `$id` variable to select the data record to be deleted. This information is then sent to the GraphQL server and the record is deleted via the resolver function. Because you're moving the queries to a file with the `.ts` extension as well as adding them, you have to adjust the `codegen.yaml` file a bit. [Listing 16.11](#) contains the corresponding changes:

```

schema: http://localhost:3001/api
documents: './src/**/*.graphql.ts'
generates:
  src/graphql/generated.ts:
    plugins:
      - typescript
      - typescript-operations
      - typescript-react-apollo
    config:
      withHooks: true
      withMutationFn: true

```

Listing 16.11 Adjustment of the codegen.yaml file

First, you change the path in the `documents` property so that the `graphql.ts` file containing the queries can be correctly located. Also, under `config`, you add the line, `withMutationFn: true`. By doing so, you make sure that Code Generator generates not only the query hooks, but also mutation hooks for you. If you now run the `npm run generate` command

again, the code generator will create the new version of the hook functions so that you can use them in your component:

```
import React from 'react';
import {
  useBooksListQuery,
  useDeleteBookMutation
} from './graphql/generated';

const List: React.FC = () => {
  const { data, loading, error } = useBooksListQuery();
  const [deleteBook] = useDeleteBookMutation(
    { refetchQueries: ['BooksList'] }
  );

  if (loading) {
    return <div>Loading...</div>;
  }

  if (error) {
    return <div>An error has occurred.</div>;
  }

  return (
    <table>
      <thead>
        <tr>
          <th>Title</th>
          <th>ISBN</th>
          <th></th>
        </tr>
      </thead>
      <tbody>
        {data?.book?.map(
          (book) =>
            book && (
              <tr key={book.id}>
                <td>{book.title}</td>
                <td>{book.isbn}</td>
                <td>
                  <button
                    onClick={() => deleteBook(
                      { variables: { id: book.id } }
                    )}>

```

```

        Delete
      </button>
    </td>
  </tr>
)
  )}
</tbody>
</table>
);
};

```

```
export default List;
```

Listing 16.12 Integration of the “deleteBook” Mutation (src/List.tsx)

Inside the `List` component in [Listing 16.12](#), you first use the `useDeleteBookMutation` hook function and pass the configuration object that will cause the query named `BooksList` to be updated. In the last step, you integrate a button to execute the `deleteBook` mutation. You pass an object with the `variables` key to the `deleteBook` function from the mutation hook, which in turn contains an object with the `id` key and the respective ID of the book record as a value. If you switch to your browser with this code state, you’ll get a view like the one shown in [Figure 16.3](#).

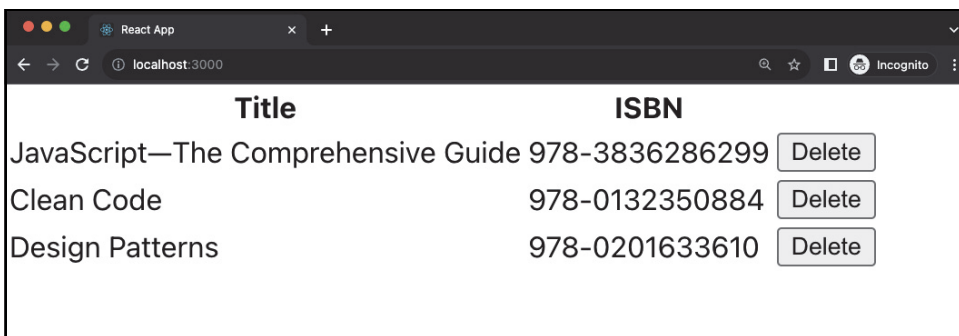


Figure 16.3 Integration of the Delete Button

If you click on the **Delete** button in a row, the corresponding data record will be deleted without further query and the view will be updated.

With the requests implemented so far, you can read data and display them in the list, as well as edit data via a mutation or, in this specific case, delete an existing data record. Editing and creating new data records works in a similar way via parameterized mutations.

In the following section, we’ll introduce *Apollo Client Devtools*, a tool for developing Apollo applications.

16.3 Apollo Client Devtools

Apollo Client Devtools is a Chrome extension, which you can install via the browser's extension manager. After successful installation, you'll see a new tab called **Apollo** in the development tools of the browser.

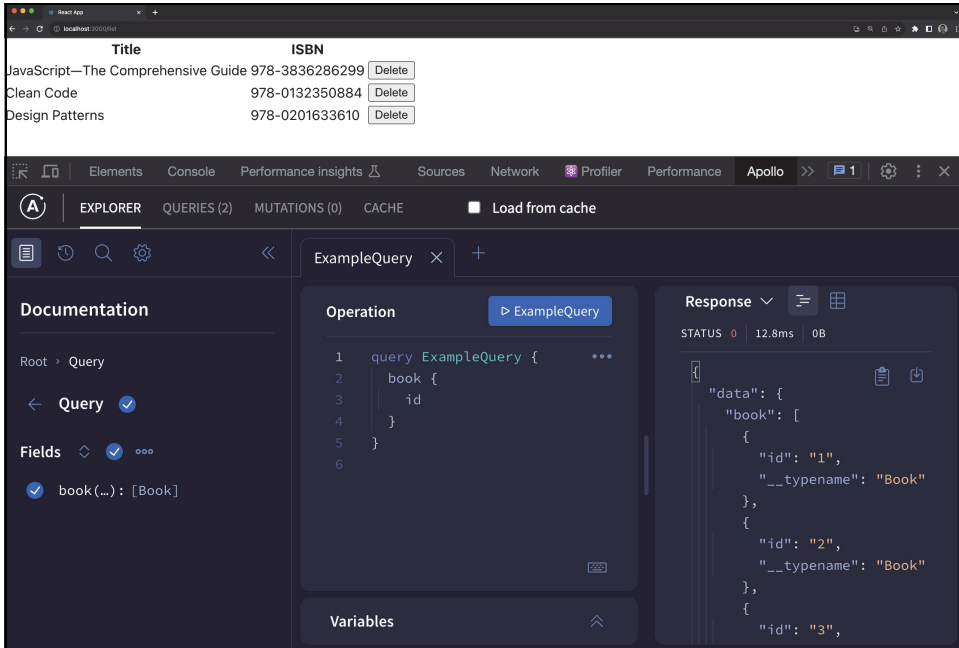


Figure 16.4 Apollo Client Devtools

In [Figure 16.4](#), you can see the view of the opened dev tools. On the left-hand side of the window, you have several options. GraphQL is open by default. The following options are available:

- **Explorer**
You can use the explorer to send any queries to the GraphQL server of your application.
- **Queries**
Using this option, you can view the queries of your application and get details like variables and the query string.
- **Mutations**
If a mutation such as deleting a record is performed, you will see the related information in the **Mutation** view. Again, you can see the variable assignment as well as the mutation string.
- **Cache**
This area reflects the current state of the Apollo cache of your application.

Apollo Client Devtools is suitable for tracking down errors and undesired behavior in connection with GraphQL and is a useful tool especially in combination with the browser's debugger.

16.4 Local State Management Using Apollo

The Apollo client is not only a means of communicating with a GraphQL server with an efficient caching mechanism, but can also be used for local state management in an application and is thus able to serve a similar purpose as Redux. In this section, you prepare to authenticate to the GraphQL interface by storing a token locally in the browser that you later send with each request to the server.

16.4.1 Initializing the Local State

Apollo provides several ways to work with a local state. You can use the so-called reactive variables, local-only fields, or local resolvers. However, the use of local resolvers is marked as deprecated and will be removed in the future.

The simpler and more lightweight variants are reactive variables, which work separate from the cache. They behave similarly to ordinary variables in JavaScript, but when updated cause the active GraphQL queries to be refreshed. In the following sections, we'll implement authentication along with reactive variables.

You should generally make sure to keep your application clean. This includes that a component should take care of only one thing if possible. For this reason, you swap out the Apollo-related content to a new file named *apolloClient.ts* in the *src* directory. The contents of this file are shown in [Listing 16.13](#):

```
import { InMemoryCache, ApolloClient, makeVar } from '@apollo/client';

const client = new ApolloClient({
  uri: 'http://localhost:3001/api',
  cache: new InMemoryCache(),
});

export const token = makeVar('');

export default client;
```

Listing 16.13 Initialization of the Apollo Client and Definition of a Reactive Variable (src/apolloClient.ts)

You use the `makeVar` function to create a reactive variable. You can directly assign an initial value to it; in this case, an empty string is the starting value. You export this reactive variable under the name `token` so that you can access it from your components.

Now you need to integrate the newly created Apollo client into the `App` component. You can see the source code required for this in [Listing 16.14](#):

```
import './App.css';
import React from 'react';
import { ApolloProvider } from '@apollo/client';
import List from './List';
import client from './apolloClient';

const App: React.FC = () => {
  return (
    <ApolloProvider client={client}>
      <List />
    </ApolloProvider>
  );
}

export default App;
```

Listing 16.14 Integration of the Apollo Client (`src/App.tsx`)

This means that the application still works as before, so you can now set about using the local state in your application.

16.4.2 Using the Local State

You get the token for accessing the GraphQL interface from the backend via the `/login` route, to which you submit the user name `admin` and password `secret` via HTTP POST. For this purpose, you implement a simple registration form located in the `Login.tsx` file in the `login` directory. You can see the source code of the component in [Listing 16.15](#):

```
import { useReactiveVar } from '@apollo/client';
import React, { ChangeEvent, FormEvent, useState } from 'react';
import { token } from './apolloClient';

const Login: React.FC = () => {
  const serverToken = useReactiveVar(token);

  const [credentials, setCredentials] = useState({
    username: '',
    password: '',
  });
};
```

```

function handleChange(event: ChangeEvent<HTMLInputElement>): void {
  setCredentials((prevCredentials) => ({
    ...prevCredentials,
    [event.target.name]: event.target.value,
  }));
}

const handleSubmit = async (event: FormEvent<HTMLFormElement>) => {
  event.preventDefault();
  const response = await fetch('http://localhost:3001/login', {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json',
    },
    body: JSON.stringify(credentials),
  });
  const data = await response.text();

  token(data);
};

if (serverToken !== '') {
  return <></>;
}

return (
  <form onSubmit={handleSubmit}>
    <div>
      <label htmlFor="username">Username: </label>
      <input
        type="text"
        value={credentials.username}
        onChange={handleChange}
        name="username"
        id="username"
      />
    </div>
    <div>
      <label htmlFor="password">Password: </label>
      <input
        type="password"
        value={credentials.password}
        onChange={handleChange}
        name="password"
      />
    </div>
  </form>
)

```

```
        id="password"
      />
    </div>
    <button type="submit">Submit</button>
  </form>
);
};

export default Login;
```

Listing 16.15 Login Form (src/Login.tsx)

At the start of the component, you use `useReactiveVar` with the reactive variable created earlier in the `apolloClient.ts` file to get a reference to the value of this variable.

Then you define a local state that takes the user name and password from the input fields. The `handleChange` function ensures that the input fields are controlled by the component and connected to the state. You use the `handleSubmit` function to make sure that the credentials are sent to the server, and then call the `token` function with the token value. This function provides write access to the reactive variable. To display the login form only if no token exists yet, you use the `serverToken` variable that contains the value of the reactive variable and display an empty fragment if a token already exists, or display the login form to allow users to log in.

Finally, you still need to integrate the `Login` component into the `App` component. [Listing 16.16](#) shows the corresponding code section:

```
import './App.css';
import React from 'react';
import { ApolloProvider } from '@apollo/client';
import List from './List';
import client from './apolloClient';
import Login from './Login';

const App: React.FC = () => {
  return (
    <ApolloProvider client={client}>
      <Login />
      <List />
    </ApolloProvider>
  );
};

export default App;
```

Listing 16.16 Integration of the “Login” Component (src/App.tsx)

Similar to the login routine, you should also proceed with the list display in this example and hide the list if there is no token yet. You can achieve this by accessing the reactive variable again in read mode as in [Listing 16.17](#) and, if no token is available yet, rendering an empty fragment:

```
...
import { useReactiveVar } from '@apollo/client';
import { token } from './apolloClient';

const List: React.FC = () => {
  ...
  const serverToken = useReactiveVar(token);
  if (!serverToken) {
    return <</>;
  }
  ...
};

export default List;
```

Listing 16.17 Hiding the List if No Token Is Present (src/List/List.tsx)

Having made these adjustments, you can reload your application and use the login form shown in [Figure 16.5](#).

Now you can indeed log in and receive a token, which will be stored correctly. However, if you enable authentication on the server side, the list will no longer be displayed correctly. That's because the server doesn't accept the request as the Authorization header has not been set correctly. We'll address this problem in the next section.

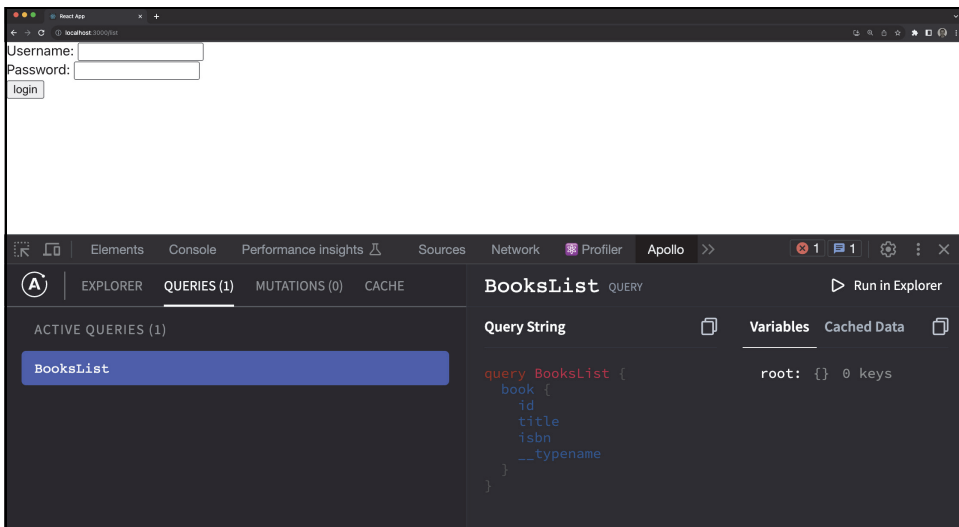


Figure 16.5 Login Form

16.5 Authentication

Because a GraphQL interface uses HTTP as its transport protocol, you can secure it just like any other web interface. In this example, we use JWT to make sure that only authorized users are allowed to query the interface. In the sample code for this book, the backend is extended so that there is a `/login` route to which you can send the user name `admin` and password `test` via HTTP POST and receive a valid token. The `/api` route is secured using middleware so that access is only possible with a valid token.

The Apollo client allows you to intercept and process all requests to the server. In the current example, you need this functionality to set the `Authorization` header correctly in the request. For this purpose, you extend the `apolloClient.ts` file as shown in [Listing 16.18](#):

```
import {
  InMemoryCache,
  ApolloClient,
  makeVar,
  createHttpLink,
} from '@apollo/client';
import { setContext } from '@apollo/client/link/context';

export const token = makeVar('');

const authLink = setContext((root, { headers }) => ({
  headers: {
    ...headers,
    authorization: `Bearer ${token()}`,
  },
}));

const client = new ApolloClient({
  link: authLink.concat(
    createHttpLink({
      uri: 'http://localhost:3001/api',
    })
  ),
  cache: new InMemoryCache(),
});

export default client;
```

Listing 16.18 Authentication Using the Apollo Client (src/apolloClient.ts)

The callback function you pass to the `setContext` function is executed on every request. Here, you also have access to the value of the reactive variable where the token resides, and you can put it in the `Authorization` header. To include the resulting Apollo link, you use the `concat` method and merge it with the existing `HttpLink`.

In the sample implementation, the `List` component is rendered in parallel with the `Login` component. This means that it won't be redisplayed as soon as the user logs in. Although Apollo makes sure that all queries based on the reactive variable are reloaded, this doesn't apply to the books list. For this reason, you must ensure that the `BooksList` query is executed again once the token has been set. You can achieve this in the simplest case by resetting the cache after setting the token. In [Listing 16.19](#), you can see how to do this using the `client.resetStore` method:

```
...
const Login: React.FC = () => {
  ...
  const handleSubmit = async (event: FormEvent<HTMLFormElement>) => {
    event.preventDefault();
    const response = await fetch('http://localhost:3001/login', {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json',
      },
      body: JSON.stringify(credentials),
    });
    const data = await response.text();
    client.resetStore();
    token(data);
  };
  ...
};

export default Login;
```

Listing 16.19 Forcing the List to Be Updated (src/Login/Login.tsx)

This adjustment now enables you to log into your application, and the code will ensure that the token is stored correctly in the reactive variable. You include the token stored in this way in the server communication via `HttpLink` so that you can also access protected resources without any problem. After the successful login and associated cache reset, Apollo reloads the books list, while React updates the display so that the list is visible.

16.6 Summary

In this chapter, you learned about GraphQL as an alternative to the commonly used REST interfaces. The difference between the two approaches is that with GraphQL you are much more flexible in your queries. You can query related data structures, the *graphs*, and don't need to send separate queries for multiple levels of objects.

This chapter covered the following points:

- The most important basic terms related to GraphQL are as follows:
 - The *schema* specifies the structure of the interface data.
 - A *query* allows read access to the interface.
 - *Mutations* allow write access to the GraphQL server.
- GraphQL queries can be parameterized and thus become dynamic.
- For client-side applications, various libraries exist that facilitate the communication with a GraphQL server. One of the most commonly used libraries is the *Apollo client*.
- You formulate queries in the Apollo client using the Hook API. The `gql` function is used in combination with a template string.
- The *GraphQL Code Generator* allows you to create typed hook functions for each query, which you can then use to send your queries.
- You can use *Apollo Client Devtools* to inspect the various aspects of your Apollo application and find any kind of bug faster.
- Not only does the Apollo client allow you to use a remote GraphQL server, it also allows you to perform centralized state management. You can implement this either by a local extension of the schema or by using the so-called reactive variables.
- *Authentication* via various mechanisms such as JWT is also possible with the Apollo client.

In the next chapter, you'll learn how to use internationalization to prepare your React application for a wider target group.

Chapter 17

Internationalization

React applications aren't just about a stable architecture and a good usable interface. The frontend should also be usable for the largest possible number of users. In such a case, the language of choice is usually English. While this allows you to potentially reach many people, only those whose native language is English are likely to feel comfortable with this solution. Consequently, for an application that you want to use internationally, a single language—even if it's English—is not an option. However, there's more to the term *internationalization*, or *I18N* for short, than the mere multilingualism of an application. The formatting of number and date values also falls within this topic. In this chapter, you'll learn how to internationalize your application to tap further user potential.

There are various approaches to internationalizing an application. For example, it's possible to deliver different applications translated into a particular language. This has the advantage that only one translation needs to be provided at a time and the links within the application can also be optimized for the respective language. But changing the language means changing the application, with all the disadvantages that entails, such as resetting the local state of the application. A more elegant solution is to switch the language within the application. For this purpose, the individual texts to be translated are marked with markers and translated according to the selected language. The disadvantage of this approach is that the translations must either already be delivered with the application or be dynamically loaded.

In this chapter, you'll get to know *react-i18next*, a library that supports the second approach presented—that is, changing the language within an application.

17.1 Using react-i18next

The *react-i18next* library is based on *i18next* and developed independently from React. You can find it on GitHub at <https://github.com/i18next/react-i18next>. The library provides a hook function that allows you to internationalize your components. Furthermore, you can use the core scope of *i18next*. Although the library isn't implemented in TypeScript, it brings its own type definitions with it, so you don't need to install an additional package. In addition to the hook function that you can use for translation, *react-i18next* also provides a higher-order component as well as a regular component. Installation is done via the `npm install i18next react-i18next` command.

The examples in this chapter are based on a new React application that you initialize via the `npx create-react-app library --template typescript` command.

Before you can use `react-i18next`, you need to configure the `i18next` base library and link them together via `react-i18next`. You save this configuration in the `i18n.ts` file directly in the `src` directory. You can see the contents of this file in [Listing 17.1](#):

```
import i18n from 'i18next';
import { initReactI18next } from 'react-i18next';
import de from './resources/de.json';
import en from './resources/en.json';

i18n.use(initReactI18next).init({
  fallbackLng: 'en',
  lng: 'en',
  interpolation: {
    escapeValue: false,
  },
  resources: {
    en: {
      translation: en,
    },
    de: {
      translation: de,
    },
  },
});

export default i18n;
```

Listing 17.1 Configuration of `react-i18next` (`src/i18n.ts`)

The `i18next` internationalization library has a modular structure. You can already see that in the configuration. You can configure the library for your application via the `i18n` object that you import from the `i18next` package. The `use` method allows you to add other plugins to the core library, such as the `react-i18next` plugin in this case, extending the functionality of the library. This plugin connects `i18next` to your React application and ensures that the function you create using the `useTranslation` hook is able to provide the appropriate translations.

The `init` method allows you to configure `i18next` yourself. In the following list, you'll find an explanation of each configuration option used:

- **fallbackLng**

You can use this option to specify which language `i18next` should use if the selected language isn't available. If you specify the value `false` here, no fallback language will be used.

- **lng**

The `lng` option allows you to set the language to use. The `i18next` library offers several ways to set the language. Initialization is one of them. Apart from that, you can also change the language at runtime.

- **interpolation/escapeValue**

The `false` value is used to turn off the escaping of `i18next`, which the library uses to prevent cross-site scripting (XSS) injections. The reason for this is that React itself already takes care of correct escaping.

- **resources**

This property enables you to pass the translations that `i18next` is supposed to use in your application. You define one property per language used, which in turn contains a `translation` property with the actual translations.

Alternatively, you can use different namespaces at this point to better divide your translations. For more information on this topic, visit www.i18next.com/principles/namespaces.

There are several ways to load the translations. The simplest one is to enter the translations directly into the `resources` property. This has the advantage that the translations are integrated directly in the bundle. However, it also has a negative impact on the bundle size. To keep the configuration file manageable, you should swap out the translations to a separate file, which you can then include via an `import` statement. The two resource files used for German and English are stored under the names *de.json* and *en.json*, respectively, in the `src/resources` directory. [Listing 17.2](#) contains the source code of the *en.json* file:

```
{
  "title": "Books list",
  "list": {
    "title": "Title",
    "isbn": "ISBN",
    "release": "Release date",
    "price": "Price"
  }
}
```

Listing 17.2 Translation File (`src/resources/en.json`)

The configuration hasn't yet been integrated into your application. But unlike other React libraries and extensions, you don't need to include `i18next` via a separate provider. In this case, it suffices to load the configuration in the initial file of your application. [Listing 17.3](#) shows the source code of the `App` component with the corresponding import. The `List` component that you're going to implement and translate in the next step has already been integrated.

```
import React from 'react';
import List from './List';
import './App.css';
import './i18n';

const App: React.FC = () => {
  return <List />;
}

export default App;
```

Listing 17.3 Import of the i18next Configuration (src/App.tsx)

With these adjustments, you've prepared your application to the point that you can translate your components. For this purpose, you define a `List` component that displays a list of books. Because the focus is on internationalization, the component is static. The data for the display is located in an array outside the component. State and effect are completely omitted in this example. [Listing 17.4](#) shows the implementation of the `List` component where the translation of the static texts has already been integrated:

```
import React from 'react';
import { useTranslation } from 'react-i18next';

const books = [
  {
    id: '1',
    title: 'JavaScript - The Comprehensive Guide',
    isbn: '978-3836286299',
    release: 1632960000000,
    price: 49.9,
  },
  {
    id: '2',
    title: 'Clean Code',
    isbn: '978-0132350884',
    release: 1217548800000,
    price: 29.09,
  },
  {
    id: '3',
    title: 'Design Patterns',
    isbn: '978-0201633610',
    release: 867715200000,
    price: 34.99,
  },
];
```

```

    },
  ];

  const List: React.FC = () => {
    const { t } = useTranslation();
    return (
      <>
        <h1>{t('title')}

```

Listing 17.4 Implementation of the “List” Component with Translation of Static Texts (src/List.tsx)

The `t` function, which you get via the `useTranslation` hook, allows you to translate the content of your component. For this purpose, you call the function with the respective key. In this case, you have a nested structure in the resource file. To address a specific translation resource, you specify the path to the respective field separated by dots. The heading is at the top level, so you can directly use the `title` key here. The elements of the table head are below the `list` key, so you use `list.title` for the title, for example.

Due to the configuration of `i18next`, your application is now able to deliver the `List` component in two languages: German and English. But at the moment, users don't have the option to change the language, so you have to do it manually via the `lng` property within the `i18next` configuration. We'll deal with this problem later in this chapter, but first you'll learn what alternatives you have for loading and including translation resources.

17.1.1 Loading Language Files from the Backend

Currently, the resource files are still a fixed part of your application and thus also have a negative impact on the bundle size. However, the `i18next` library allows you to load the resources from a web server as well. To do this, you must first install the `i18next-http-backend` extension via the `npm install i18next-http-backend` command. Then you configure this plugin in the `i18n.ts` file of your application. You can see the source code required for this in [Listing 17.5](#):

```
import i18n from 'i18next';
import { initReactI18next } from 'react-i18next';
import HttpApi from 'i18next-http-backend';

i18n
  .use(initReactI18next)
  .use(HttpApi)
  .init({
    fallbackLng: 'en',
    lng: 'en',
    interpolation: {
      escapeValue: false,
    },
    backend: { loadPath: '/locales/{{lng}}.json' },
  });

export default i18n;
```

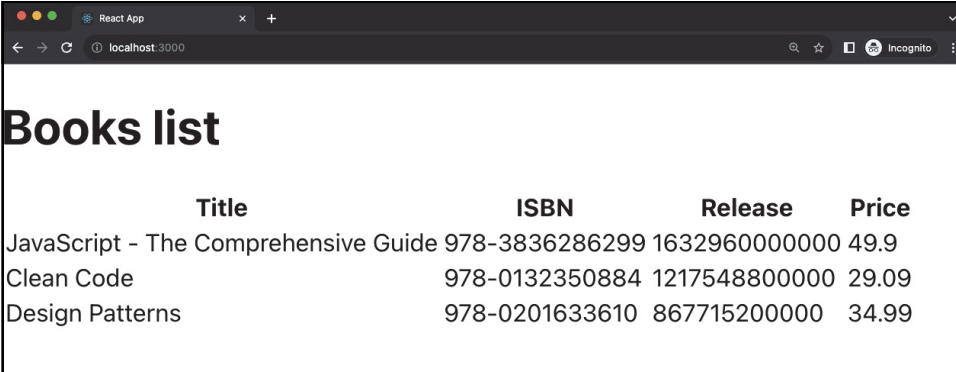
Listing 17.5 Integration of the “`i18next-http-backend`” Plugin (`src/i18n.ts`)

For the integration, you import the plugin as `HttpApi` and include it via the `use` method of `i18next`. In the configuration object of the `init` method, you can further configure the plugin with the `backend` key and specify, for example, that the resource files are to be found in the `locales` path and are named according to the respective language.

To make your application work in development mode, you move the two resource files to a new `locales` directory that you create in the `public` directory of the application. After these adjustments, you can reload your application and see that it's translated correctly. When you change the `lng` property in the configuration from `en` to `de`, you'll see

that the German translation is used. In the **Network** tab of your browser's developer tools, you'll see that there is one request to the server per language used. In the browser, you should see a view like the one shown in [Figure 17.1](#).

The current implementation doesn't allow the users of your application to change the language. If you've set English as default, your application will always be provided in English. In the next step, you'll have the application translated into the language currently used in the browser.



Title	ISBN	Release	Price
JavaScript - The Comprehensive Guide	978-3836286299	16329600000000	49.9
Clean Code	978-0132350884	12175488000000	29.09
Design Patterns	978-0201633610	8677152000000	34.99

Figure 17.1 Translated Books List

17.1.2 Using the Language of the Browser

The `react-i18next` plugin is by far not the only extension for `i18next`. The library provides solutions for numerous tasks in the form of plugins. This also includes language recognition. For this purpose, you install the `i18next-browser-languagedetector` plugin via the `npm install i18next-browser-languagedetector` command. Then you include the plugin in the `i18n.ts` file. The updated source code of this file is shown in [Listing 17.6](#):

```
import i18n from 'i18next';
import { initReactI18next } from 'react-i18next';
import HttpApi from 'i18next-http-backend';
import LanguageDetector from 'i18next-browser-languagedetector';

i18n
  .use(initReactI18next)
  .use(HttpApi)
  .use(LanguageDetector)
  .init({
    debug: true,
    fallbackLng: 'en',
    interpolation: {
      escapeValue: false,
    },
  });
```



```
    backend: { loadPath: '/locales/{{lng}}.json' },  
  });
```

```
export default i18n;
```

Listing 17.6 Integration of the Language Detection Plugin (src/i18n.ts)

The language detector analyzes the browser language using the `navigator.language` property and sets the appropriate value. If no translations are available for the language, `i18next` uses the fallback language—in this case, English. Besides the browser language, the language detector supports other language selection options. For example, you can use the `lng=de` query string to switch to German. The URL in this case is then `http://localhost:3000/?lng=de`.

This extension gives you some flexibility, but the options provided by the language detector aren't really user-friendly. Also, it's not really useful to adjust the configuration of the entire browser for the language switch in the application, and using the query string for the switch is also out of the question. For this reason, you'll provide your users with a way to switch the language dynamically at runtime in the next step.

17.1.3 Extending the Navigation with Language Switching

Once your application has been loaded, you give your users the option to choose the language. For this purpose, you integrate one button for German and one for English. As soon as one of the buttons is clicked on, the language of `i18next` changes and the application is translated accordingly. You encapsulate the ability to change the language in a separate component called `LanguageSwitch`, which you place in the `LanguageSwitch.ts` file in the `src` directory. The source code of this file is shown in [Listing 17.7](#):

```
import React from 'react';  
import { useTranslation } from 'react-i18next';  
  
const LanguageSwitch: React.FC = () => {  
  const { i18n } = useTranslation();  
  
  return (  
    <div style={{ textAlign: 'right', paddingTop: 5,  
      paddingRight: 30 }}>  
      <button  
        onClick={() => i18n.changeLanguage('de')}  
        disabled={i18n.language === 'de'}  
      >  
        DE  
      </button>  
      <button
```

```

      onClick={() => i18n.changeLanguage('en')}
      disabled={i18n.language === 'en'}
    >
      EN
    </button>
  </div>
);
};

```

```
export default LanguageSwitch;
```

Listing 17.7 Implementation of the “LanguageSwitch” Component (src/LanguageSwitch.tsx)

The object that the `useTranslation` function returns contains not only the `t` function but also a reference to the `i18n` instance. This in turn allows you to change the language on the fly via the `changeLanguage` method. The `LanguageSwitch` component has two button elements, in each of whose click handlers you execute the `changeLanguage` method. You can use the `language` property of the `i18n` object to find which language is currently active. You use this information to disable the button for the current language. Then you can integrate the component in the `App` component of your application to provide your users with a convenient way to switch languages. [Listing 17.8](#) contains the adjusted source code of the `App` component:

```

import React from 'react';
import List from './List';
import './App.css';
import './i18n';
import LanguageSwitch from './LanguageSwitch';

const App: React.FC = () => {
  return (
    <>
      <LanguageSwitch />
      <List />
    </>
  );
};

```

```
export default App;
```

Listing 17.8 Integration of the “LanguageSwitch” Component (src/App.tsx)

If you now switch to your browser, you can change the language between German and English as you wish. The translated strings will be exchanged dynamically, but the page won’t be reloaded. [Figure 17.2](#) shows the current state of the application interface.

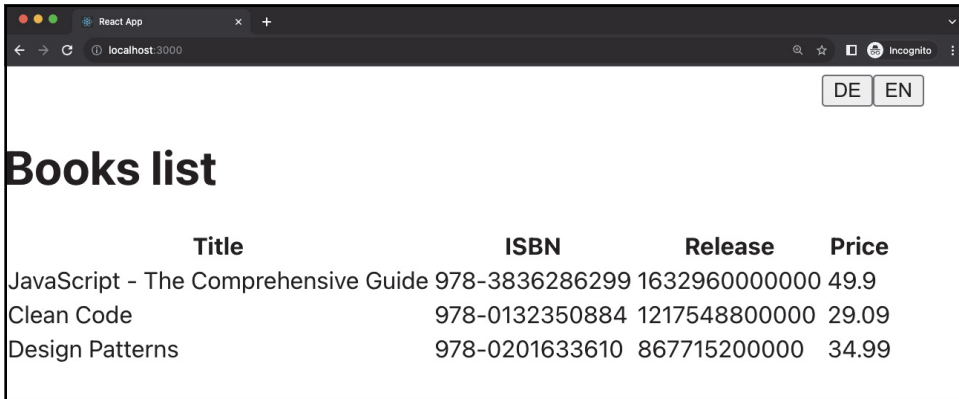


Figure 17.2 Buttons to Change the Language

17.2 Using Placeholders

Up to this point, you’ve used `react-i18next` only in the context of static texts. But in a modern single-page application, it often happens that you need to include dynamic values in such a text. For this purpose, `react-i18next` supports placeholders in translatable texts.

To demonstrate this feature, you’ll display the number of pages of the books in the books list and add the “Page” string at the end. In the first step, you add a new `pages` property to the data in the `List` component, and then you add another column to the table in the component for displaying the number of pages. [Listing 17.9](#) shows the adjusted component:

```
import React from 'react';
import { useTranslation } from 'react-i18next';

const books = [
  {
    id: '1',
    title: 'JavaScript - The Comprehensive Guide',
    isbn: '978-3836286299',
    release: 1632960000000,
    price: 49.9,
    pages: 1273,
  },
  {
    id: '2',
    title: 'Clean Code',
    isbn: '978-0132350884',
    release: 1217548800000,
    price: 29.09,
```

```

    pages: 464,
  },
  {
    id: '3',
    title: 'Design Patterns',
    isbn: '978-0201633610',
    release: 867715200000,
    price: 34.99,
    pages: 395,
  },
];

const List: React.FC = () => {
  const { t } = useTranslation();
  return (
    <>
      <h1>{t('title')}

```

Listing 17.9 Extension of the “List” Component to Display the Number of Pages (src/List.tsx)

The interesting part of this code example is the last call of the `t` function. Instead of the key, you pass the key and an object to this function. The object contains the values for the placeholders you define in the resource. This is done by means of two pairs of curly brackets, as shown in [Listing 17.10](#) for the example of the English translation:

```
{
  "title": "Books list",
  "list": {
    "title": "Title",
    "isbn": "ISBN",
    "release": "Release date",
    "price": "Price",
    "pages": "Pages",
    "pagesValue": "{{pages}} pages"
  }
}
```

Listing 17.10 Adding the Number of Pages to the Translation File (public/locales/en.json)

For the translation to work without problems, you need to add the `pages` and `pagesValue` keys in both translation files. The translation of `pagesValue` contains the placeholder `pages`, which `i18next` then replaces with the passed value. Using placeholders has the advantage that the placeholder can be in different places in different languages, and it is thus much more flexible than if you used string concatenation. When you switch to the browser, you should see a view like the one shown in [Figure 17.3](#).

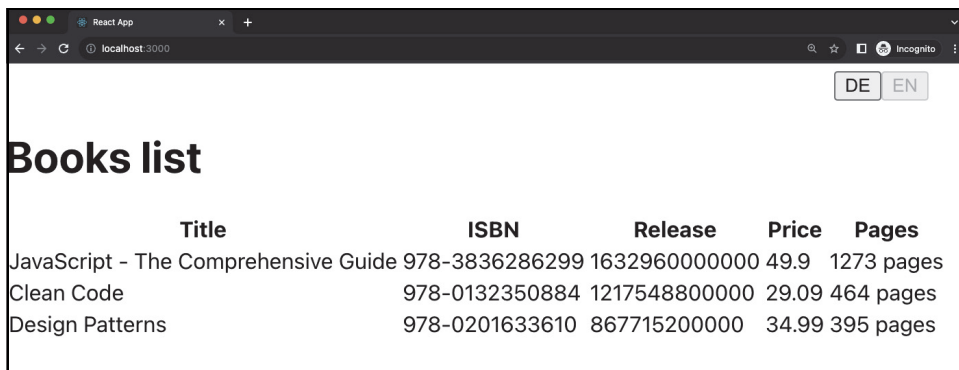


Figure 17.3 Translation with Placeholders

17.3 Formatting Values

The internationalization of an application consists not only of translation of strings, but also of adaptations of numbers and dates. The `i18next` library provides several ways to deal with such values. Basically, the library is based on the interfaces of the `Intl`

interface of the browser. You can use this interface to format numbers, currencies, and date values, for example.

17.3.1 Formatting Numbers and Currencies

In the first step, you format the prices of the books as regular numbers. For this purpose, you define a new key as shown in [Listing 17.11](#), to which you pass the price of the book, and `i18next` then takes care of the correct formatting based on its configuration:

```
import React from 'react';
import { useTranslation } from 'react-i18next';

const books = [...];

const List: React.FC = () => {
  const { t } = useTranslation();
  return (
    <>
      <h1>{t('title')}</h1>
      <table>
        <thead>...</thead>
        <tbody>
          {books.map((book) => (
            <tr key={book.id}>
              <td>{book.title}</td>
              <td>{book.isbn}</td>
              <td>{book.release}</td>
              <td>{t('list.priceValue', { price: book.price })}</td>
              <td>{t('list.pagesValue', { pages: book.pages })}</td>
            </tr>
          ))}
        </tbody>
      </table>
    </>
  );
};

export default List;
```

Listing 17.11 Formatting the Prices (src/List.tsx)

The component only needs to handle using the appropriate key and passing the value. You implement the formatting in the translation resource. [Listing 17.12](#) shows the German translation file with the number formatting as an example:

```
{
  "title": "Books list",
  "list": {
    ...
    "priceValue": "{{price, number(minimumFractionDigits: 2)}}"
  }
}
```

Listing 17.12 Number Formatting in the Translation File (public/locales/en.json)

To format the number, you use the interpolation notation again so that `i18next` can access the `price` variable that you passed to the `t` function in the component. In addition to the variable, you pass the formatting option you want to use. In this case, we're talking about formatting numbers, so you want to use `number`. You can pass additional options to this formatting, such as `minimumFractionDigits: 2` to always force the display of two decimal places.

Not only does the internationalization of `i18next` ensure the correct display of decimal places, but it also uses the appropriate decimal separator. For English, it's a period; for German, that's a comma. If you look at your application with this change in the German translation, you'll therefore see the value 49,9 instead of value 49.9 in the first line.

Because prices aren't ordinary number values, but currencies, you can go one step further and use `currency` formatting instead of `number` formatting. In [Listing 17.13](#), you format the `price` value in the translation file as a dollar value:

```
{
  "title": "Books list",
  "list": {
    ...
    "priceValue": "{{price, currency(USD)}}"
  }
}
```

Listing 17.13 Formatting the Price as a Dollar Value (public/locales/en.json)

The `currency` formatting with the `EUR` value ensures that `i18next` uses the correct decimal separator for the selected language and also sets the currency symbol—in this case, the € sign. Note that the internationalization interface only takes care of the formatting, not the conversion of values. So if you want to use dollar values in English instead of euros, you'll have to take care of the conversion yourself.

For more information on the internationalization of numbers, visit <http://s-prs.co/v570507>.

17.3.2 Formatting Date Values

In the current implementation of the `List` component, you display the timestamp of the current date as the publication date. For normal users, this numerical value has very little significance. For this reason, you should ensure that the application outputs the date in a format that can be read by humans. For this purpose, you also use internationalization with `i18next` as dates are displayed differently in different languages.

Unlike number and currency formatting, with date formatting you specify the information about the format in the component using an object structure. The `formatParams` object represents an alternative to the previously presented passing of formatting options in the translation resource. In this case, you can better influence the formatting in the source code if it becomes necessary. You also move the complexity from the translation file to the component.

The source code in [Listing 17.14](#) makes sure that both the day and the month are displayed in two-digit forms and the year in four digits for the publication date:

```
import React from 'react';
import { useTranslation } from 'react-i18next';

const books = [...];

const List: React.FC = () => {
  const { t } = useTranslation();
  return (
    <>
      <h1>{t('title')}</h1>
      <table>
        <thead>...</thead>
        <tbody>
          {books.map((book) => (
            <tr key={book.id}>
              <td>{book.title}</td>
              <td>{book.isbn}</td>
              <td>
                {t('list.releaseValue', {
                  release: new Date(book.release),
                  formatParams: {
                    release: {
                      year: 'numeric',
                      month: '2-digit',
                      day: '2-digit',
                    },
                  },
                })}
              </td>
            </tr>
          ))}
        </tbody>
      </table>
    </>
  );
}
```



```

        </td>
        <td>{{t('list.priceValue', { price: book.price })}}</td>
        <td>{{t('list.pagesValue', { pages: book.pages })}}</td>
      </tr>
    )}}
  </tbody>
</table>
</>
);
};

```

```
export default List;
```

Listing 17.14 Formatting of Dates (src/List.tsx)

Because you set the format options in the component, the translation file becomes much simpler, as you can see in [Listing 17.15](#) for the German translation example:

```

{
  "Titel": "Bücherliste",
  "Liste": {
    ...
    "releaseValue": "{{Release, datetime}}"
  }
}

```

Listing 17.15 Translation of the Publication Date (public/locales/de.json)

If you look at the browser for the state of the source code now, you'll see that the entire component is correctly translated into German and English and that the numerical and time values are also correctly formatted. The result should look as shown in [Figure 17.4](#).

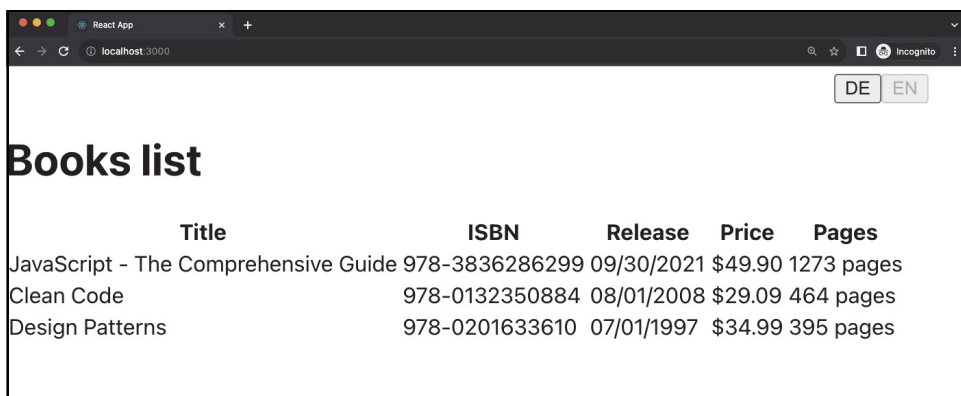


Figure 17.4 Translated View of the Books List

17.4 Singular and Plural

You probably know of applications that use data tables and display the number of data records found as information for their users. Sometimes developers don't spend much attention on that, leading to displays such as **1 data records found**. The `react-i18next` library provides a solution for this problem, allowing you to display the correct information in both a monolingual and multilingual application. However, the library supports not only singular and plural, but also gradations made in some languages, such as Russian.

The `i18next` library supports singular and plural by appending an underscore and a specific key to a key. When you call the `t` function, you pass the `count` variable, which ensures that the appropriate translation will be used. In [Table 17.1](#), you can see the supported keys.

Key	Meaning
other	Fallback if no suitable property was found for the specified number.
zero	The passed value has the value 0.
singular	This translation takes effect in the case of the value 1.
two	The two property allows you to specify a translation for the value 2.
few	This translation applies in Russian, for example, for the values 2 to 4.
many	This value usually applies to the regular plural.

Table 17.1 Keys for Singular and Plural

If you need more information about plural rules, you can find it at www.unicode.org/cldr/charts/latest/supplemental/language_plural_rules.html.

To demonstrate this feature, you extend the `List` component with a search field that enables you to filter the books list. In information text, the component shows how many data records are currently being displayed. The corresponding source code of the `List` component is shown in [Listing 17.16](#):

```
import React, { useState } from 'react';
import { useTranslation } from 'react-i18next';

const books = [...];

const List: React.FC = () => {
  const [filter, setFilter] = useState('');

  const filteredBooks = books.filter((book) =>
    book.title.toLowerCase().includes(filter.toLowerCase())
```

```
);

const { t } = useTranslation();
return (
  <>
    <h1>{t('title')}</h1>

    <div>
      {t('filter')}
      <input
        type="text"
        value={filter}
        onChange={(e) => setFilter(e.target.value)}
      />
      {t('filterResults', { count: filteredBooks.length })}
    </div>

    <table>
      <thead>...</thead>
      <tbody>
        {filteredBooks.map((book) => (
          <tr key={book.id}>...</tr>
        ))}
      </tbody>
    </table>
  </>
);
};

export default List;
```

Listing 17.16 List with Filter Function (src/List.tsx)

For the filter function of the list, you first define a local state in the component where you store the current filter. Using this filter, you reduce the `books` array and show only the books whose titles contain the filter criterion. In the JSX structure of the component, you integrate an input field and the information text that indicates how many books the current filter applies to. You can translate both the label that appears before the input field and the information text. For the information text, you pass the number of hits as a variable to the translation via the `count` property. [Listing 17.17](#) shows the English translation file as an example of singular and plural:

```
{
  "title": "Books list",
  "filter": "Search: ",
```

```

"filterResults_one": "{{count}} book found",
"filterResults_other": "{{count}} books found",
"list": {...}
}

```

Listing 17.17 Singular and Plural in the Translation File (public/locales/en.json)

For the `filterResults` translation, you append the `_one` key if only one book was found. In all other cases, `i18next` uses the `_other` key and displays the appropriate translation. Again, you can check the result in the browser. For a filter that returns only one result, you should get a view like the one shown in [Figure 17.5](#).

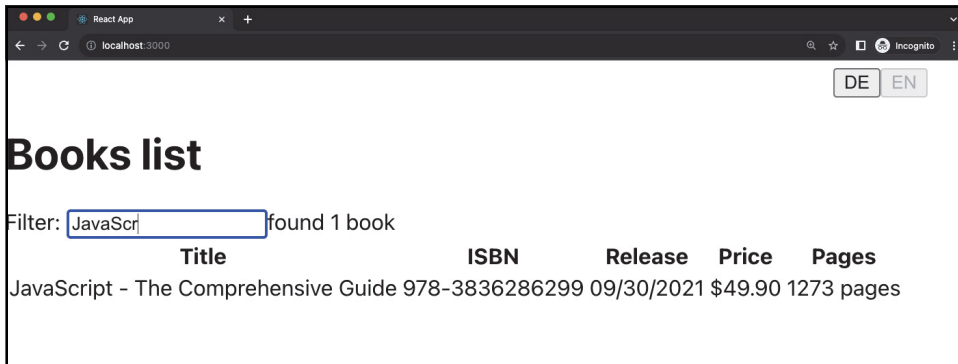


Figure 17.5 Translated Filter Function

17.5 Summary

In this chapter, you've learned how to prepare your application for the international market using `react-i18next`. It's important to note that the internationalization of an application doesn't end with the translation of content, but also involves the formatting of numbers and dates, for example. You learned about the following topics:

- The `react-i18next` library is based on the `i18next` library. This means that you need to install both packages for use in React.
- You can either specify the translations directly in the code or swap them out to separate files.
- The `react-i18next` library provides various options to internationalize your components. The `useTranslation` hook function is used most often.
- You can use the `changeLanguage` method of the `i18n` object to switch the language while your application is running.
- The `react-i18next` library allows you to translate strings and to format numbers and dates.

- Due to support of pluralization, it's possible to output different texts based on a given numerical value. Here, not only singular and plural but also finer gradations are supported.

In the following chapter, you'll learn how to use server-side rendering to further improve the performance of your application and optimize the initial loading process.

Chapter 18

Universal React Apps with Server-Side Rendering

One reason why single-page applications (SPAs) are becoming increasingly popular and a serious competitor to both traditional websites and native desktop and mobile apps is that they feel like their native counterparts when used, but bring all the benefits of a web app. It's precisely the smooth usage of such an application that stands out, and it's achieved by loading only the difference from the previous state after the initial loading process, and that in the form of pure data. The remaining structures such as HTML, CSS, and JavaScript are already in the client. Single-page applications open up entirely new avenues for web developers in terms of design and user experience. Without the page changes in the navigation, you can use animations at the state transition and inform the user about progress. However, this approach doesn't only have advantages.

One disadvantage is that the initial loading process takes a comparatively long time. First, the browser loads an almost empty HTML file, which forms the basis for the SPA and is responsible for loading all other resources. The actual application consists of a set of JavaScript, CSS, and various media files. After this initial loading process, which loads only the static data that is the same for all users, the dynamic data that the application presents must still be loaded. This process takes place asynchronously, so that the viewers of the application (and these do not necessarily have to be humans, but can also be bots of a search engine) initially see an empty page. This is followed by an empty application framework, and not until after all the data has been loaded is the application fully built. This process usually takes only a few seconds, depending on the application, but even this small amount of time can be detrimental—so when developing, you should strive to minimize the amount of time it takes for users to use your application. You can achieve this by keeping the size of your application small—that is, by delivering as little source code as possible. This limits the use of libraries, but also the overall functionality of the application, which is often not an option.

Another possible solution is to use *server-side rendering* (SSR). In this context, you shorten the process of the initial buildup of the application and do not deliver an empty HTML file first, but instead prepare it already in such a way that the browser can display the application including all display data directly after loading the initial resources. This is where the name of this concept comes from: it means that you perform the rendering process on the *server side* and then deliver the *rendered* application to the client. In this chapter, you'll learn how to use SSR to improve the performance of your application.

Universal Apps

In the context of SSR, you'll sooner or later come across the term *universal apps*. It refers to applications that run on both the client and the server. This is a basic requirement for SSR as the server has to render the application for the client.

React itself doesn't provide a prebuilt solution for SSR, just the basic functionality that lets you render the application both on the client side and on the server side, for example, or the `hydrateRoot` function that lets you control a prerendered application.

18.1 How Does Server-Side Rendering Work?

SSR is a multistep process that adds some complexity to your application. It's best explained using an overview chart such as the one in [Figure 18.1](#).

As is usually the case on the web, the initiative comes from the client. Users call the application using their browser. The client then sends a request to the server **1**. On the server side, an endpoint exists for this initial request, so the request is received correctly **2**. Within this endpoint, the SSR is implemented **3**.

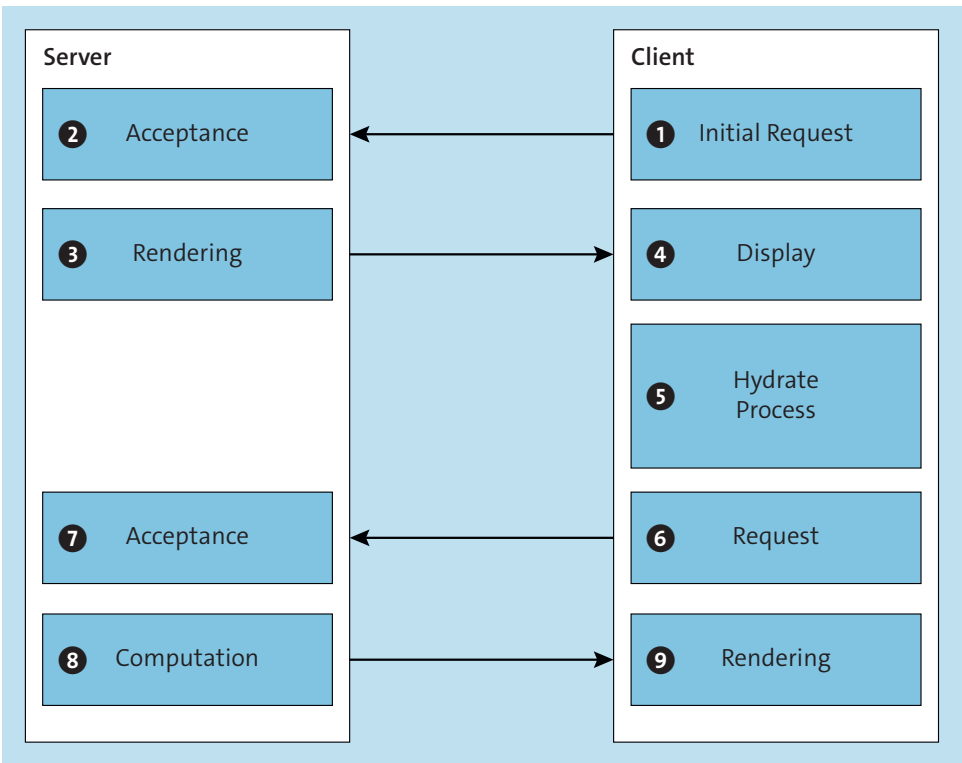


Figure 18.1 SSR Process

Here, the frontend application is executed and rendered on the server side. The result is a static version of the React application in the form of HTML. This HTML code is included in an HTML page template and sent back to the client as a response. The client initially exposes this static version ❹ and executes the source code supplied by the server. There, instead of the usual `render` function from the `react-dom` package, the `hydrateRoot` function is executed ❺. This function ensures that React takes control of the static HTML. Once this process is complete, the application runs as usual in the browser and users can interact with it. Both the hydrate process and the interaction can lead to further server communication ❻.

In the hydrate process, server communication is typically triggered by component life-cycle hooks, and during the interaction, implementations of event handlers trigger the communication. On the server side, an individual endpoint must exist for each of these requests ❼. This endpoint calculates the answer by either running an algorithm or communicating with a database ❸. The result is usually sent as a JSON structure to the client, which leads to a re-rendering of certain components there ❹.

18.2 Implementing Server-Side Rendering

In this chapter, you'll implement an SSR application yourself, step by step. Although there are frameworks like Next.js that can do the configuration for you, it's important that you understand the processes, so it's useful to take a behind-the-scenes look at SSR. In the next sections, you'll learn about all the components involved in the SSR process and implement them in a sample application. In this application, you implement the display of a list of records, which contains the main use cases that you can cover with SSR. These use cases are the initial representation of the data as well as the transfer of responsibility to React.

18.2.1 Initializing and Configuring the Server Application

Before you can start developing your application, you need to make some preparations. This is mainly because in SSR you are trying to run React in a different environment—namely, in Node.js. In the browser, Create React App configures the build process for you, but on the server side, you must do that yourself.

Initializing and Installing Dependencies

The first step is to create a new directory for your application. This directory contains the source code of the server on the top level and the source code of the frontend application in a subdirectory. In this root directory, you use the `npm init -y` command to have a `package.json` file created for your application. Then, using the `npm install @babel/core @babel/preset-env @babel/preset-react babel-loader css-loader express react react-dom webpack webpack-cli webpack-node-externals` command, you install

the packages required for the SSR. The *package.json* file that results from these two commands is shown in [Listing 18.1](#):

```
{
  "name": "ssr",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "@babel/core": "^7.18.6",
    "@babel/preset-env": "^7.18.6",
    "@babel/preset-react": "^7.18.6",
    "babel-loader": "^8.2.5",
    "css-loader": "^6.7.1",
    "express": "^4.18.1",
    "react": "^18.2.0",
    "react-dom": "^18.2.0",
    "webpack": "^5.73.0",
    "webpack-cli": "^4.10.0",
    "webpack-node-externals": "^3.0.0"
  }
}
```

Listing 18.1 Initial Version of the Server-Side *package.json* File

Via the `install` command, you've installed Babel and the plugins required for the SSR, and you've also installed the Webpack bundler. Both tools are required because you need to interpret and convert JSX on the server side. If you try to render React on the server side without any other tools, Node.js will interrupt this process very quickly due to syntax errors. So you need the intermediate step of translating the JSX code into valid Node.js code. In addition to these tools, you've also installed React and React-DOM as well as the `express` package, which allows you to implement a full-fledged web server in Node.js in a few steps.

Configuring Webpack

In the next step, you configure Webpack. To do this, you create a new file named *webpack.config.js* in the root directory of your application.

[Listing 18.2](#) contains the source code of this file:

```

const nodeExternals = require('webpack-node-externals');
const path = require('path');

module.exports = {
  mode: 'production',
  target: 'node',
  entry: './index.js',
  output: {
    path: path.join(__dirname, './build'),
    filename: 'server.bundle.js',
  },
  externals: [nodeExternals()],
  module: {
    rules: [
      {
        test: /\.js?$/,
        loader: 'babel-loader',
        exclude: /node_modules/,
      },
      {
        test: /\.css$/,
        use: [
          {
            loader: 'css-loader',
            options: {
              modules: {
                exportOnlyLocals: true,
                exportLocalsConvention: 'camelCase',
                localIdentName: '[local]_[hash:base64:5]',
              },
            },
          },
        ],
      },
    ],
  },
};

```

Listing 18.2 Configuration for the Server-Side Webpack Installation (webpack.config.js)

Webpack is a very powerful tool that allows you to cover almost all aspects of a JavaScript build process. One of the biggest drawbacks of Webpack is the complexity of its configuration. The more tasks you assign to Webpack, the more extensive the configuration becomes. To avoid making our example unnecessarily extensive, we'll omit all extensions that aren't directly related to the SSR at this point.

The configuration of Webpack consists of an object that you export using `module.exports`. The individual properties have the following meaning:

- **mode**

The `mode` property allows you to determine the mode in which Webpack is to work. Possible values are `none`, `development` and `production`. In the case of `production`, the `NODE_ENV` environment variable is set to `production` and the optimizations for the production environment are activated in plugins.

- **target**

Webpack supports various environments. Using the `target` property, you can specify the target of the build process, in our case `node` for Node.js.

- **entry**

The `entry` property sets the entry point into the application. In the example, that's the `index.js` file you will create next.

- **output**

Webpack generates new source code from a code base and configuration. You must save this in your file system to be able to run it. Select the `server.bundle.js` file in the `build` directory as the destination. If this file doesn't exist at the beginning of the build process, then Webpack will create it for you.

- **externals**

The `externals` property allows you to specify files and directories that shouldn't be included in the bundle. The `webpackNodeExternals` package helps you to ignore any packages in the `node_modules` directory, as Node.js will find them on its own during execution.

- **module**

Webpack has a modular structure. This means that the core of the tool is kept light-weight and can be extended through an interface. You can use the `module.rules` property to define rules for your build process, allowing you to include and configure `babel-loader` and `css-loader`, for example.

Configuring Babel

In the next step, you configure *Babel*, a JavaScript compiler that can translate TypeScript or JSX source code into valid JavaScript source code. Like Webpack, Babel has a modular structure. For example, Babel allows you to translate TypeScript to JavaScript or, as in our case, JSX to JavaScript. You save the configuration from [Listing 18.3](#) in the root directory of your application under the name `babel.config.js`.

```
module.exports = function (api) {  
  api.cache(true);  
  return {  
    presets: [  
      '@babel/preset-env',  
    ],  
  };  
};
```

```

    ['@babel/preset-react', { runtime: 'automatic' }]],
  ],
};
};

```

Listing 18.3 Babel Configuration (babel.config.js)

You use the `cache` method to make sure that Babel needs to execute the configuration function only once, writes the result to the cache, and then uses this cached value.

Babel uses so-called presets. These are groups of plugins or ready-made standard configurations. In the example, you use `preset-env`, which allows you to compile modern ECMAScript, and `preset-react`, which allows you to translate JSX.

Implementing the Server

With this configuration, you can move on to implementing the server process in the next step. You implement it as an express application. The source code required for this is shown in [Listing 18.4](#):

```

import { readFileSync } from 'node:fs';
import express from 'express';
import React from 'react';
import ReactDOMServer from 'react-dom/server';
import App from './frontend/src/App';

const port = 3000;
const server = express();

server.get('/', (request, response) => {
  const app = ReactDOMServer.renderToString(<App />);

  const fileContent = readFileSync(
    './frontend/build/index.html',
    'utf-8'
  ).replace('<div id="root"></div>', `<div id="root">${app}</div>`);
  response.send(fileContent);
});

server.use(express.static('./frontend/build'));

server.listen(port, () => {
  console.log(`Server is listening to http://localhost:${port}`);
});

```

Listing 18.4 Server Implementation for the SSR (index.js)

You use the `express` function to create a new instance that represents your server process. You use the `get` method of this server instance to specify that you want to bind a callback function for a `get` request to a specific path; here, this is `/`. This means that when you call `http://localhost:3000/` in the browser, Node.js executes this function and thus generates the response.

You use the `renderToString` method to render the specified `App` component into a JavaScript string instead of a DOM structure. Then you use the `readFileSync` function from the `fs` module of Node.js to read the `index.html` file from your React application. At this point, neither the `App` component nor the `index.html` file exists. However, both elements are regular structures; you know them from applications you've created using Create React App.

As you know, the `index.html` file contains an empty `div` element into which React renders your application. You now take over this step yourself by replacing the `div` element and placing a `div` element with the rendered app component there. You send this structure to the requesting client using the `response.send` method.

Then you use the `server.use` method to deliver all the remaining static assets of your application—that is, all the other HTML, JavaScript, CSS, and media files needed by your application. Last but not least, you bind the server instance to TCP port 3000 and output a success message to the console once that's done.

NPM Scripts

Your application isn't ready for SSR at this point, but you've almost completed the server-side part. You can now call Webpack and run the result using Node.js. The result will be a series of error messages because the source code of the frontend doesn't exist yet. However, so that you can comfortably start your application via `npm start` in the future, you add two more scripts to your `package.json` file:

```
{
  "name": "ssr",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "prestart": "webpack --config webpack.config.js",
    "start": "node build/server.bundle.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {...}
}
```

Listing 18.5 Extending the `package.json` File with Additional NPM Scripts (`package.json`)

The two scripts from [Listing 18.5](#) enable you to run the `npm start` command, which causes the `prestart` script to run Webpack first and hence Babel. Node.js then uses the result to start the server process.

18.2.2 Implementing the Client-Side Application

For the client side, you create a new React application in the root directory of the application using the `npx create-react-app frontend` command. In this application, you implement a new component named `List` in the `src` directory. The source code of this component is shown in [Listing 18.6](#):

```
const books = [
  {
    id: '1',
    title: 'JavaScript - The Comprehensive Guide',
    isbn: '978-3836286299',
  },
  {
    id: '2',
    title: 'Clean Code',
    isbn: '978-0132350884',
  },
  {
    id: '3',
    title: 'Design Patterns',
    isbn: '978-0201633610',
  },
];

function List() {
  return (
    <table>
      <thead>
        <tr>
          <th>Title</th>
          <th>ISBN</th>
        </tr>
      </thead>
      <tbody>
        {books.map((book) => (
          <tr key={book.id}>
            <td>{book.title}</td>
            <td>{book.isbn}</td>
          </tr>
        ))}
      </tbody>
    </table>
  );
}
```

```
        </tbody>
      </table>
    );
  }

  export default List;
```

Listing 18.6 Implementation of the “List” Component (frontend/src/List.js)

You integrate the List component in the App component of your frontend, as shown in [Listing 18.7](#):

```
import './App.css';
import List from './List';

function App() {
  return <List />;
}

export default App;
```

Listing 18.7 Integration of the “List” Component into the Application (frontend/src/App.js)

This code state allows you to render your application on the server side. What’s still missing is the transfer of control to React—that is, the *hydration* process. You can activate this process using the `hydrateRoot` function from `react-dom/client`. This function call replaces the `createRoot` call in the `index.js` file of your application. [Listing 18.8](#) shows the adjusted source code of the `index.js` file:

```
import React from 'react';
import './index.css';
import App from './App';
import reportWebVitals from './reportWebVitals';
import { hydrateRoot } from 'react-dom/client';

const container = document.getElementById('root');
const root = hydrateRoot(
  container,
  <React.StrictMode>
    <App />
  </React.StrictMode>
);

reportWebVitals();
```

Listing 18.8 Calling the “hydrateRoot” Function (frontend/src/index.js)

The final step now is to build your application for delivery using the `npm run build` command in the *frontend* directory, thus preparing all assets and storing them in the *build* directory. Once you've successfully executed this command, you can switch back to the root directory and start the server process there via the `npm start` command.

Then you can switch to the browser and open `http://localhost:3000/`. You should see a list of books. However, the real excitement is hidden in the developer tools. If you display the source code of the initial resource there, you'll see that not an empty div is delivered, but the prerendered application, as shown in [Figure 18.2](#).

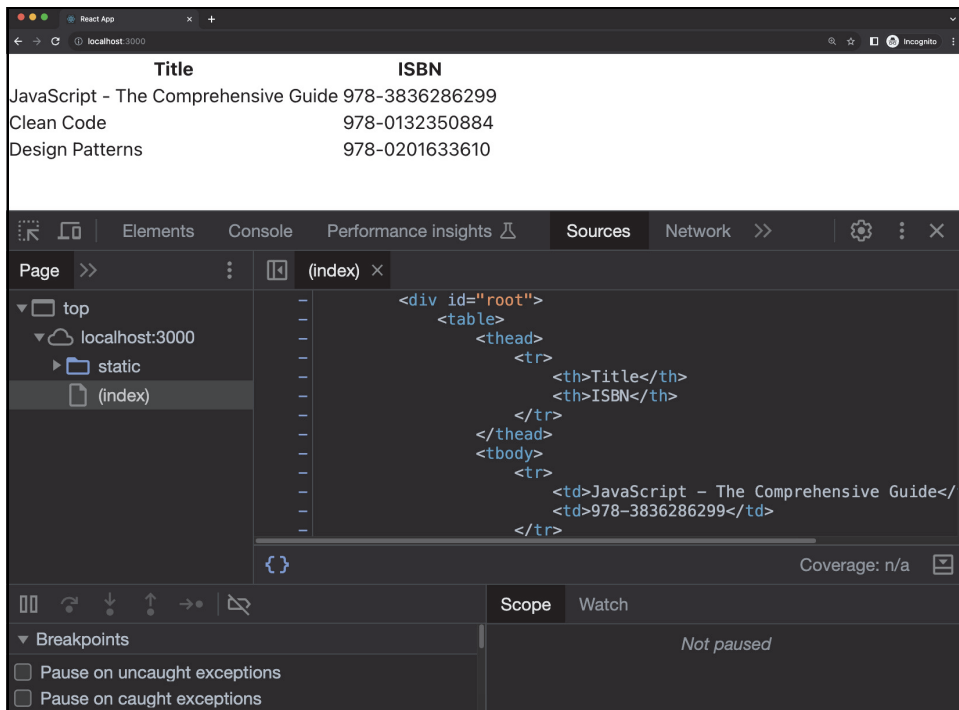


Figure 18.2 Delivery of the Prerendered Component

18.2.3 Dynamics in Server-Side Rendering

In the current implementation, you render the `List` component statically, but it has no dynamics. The data that's displayed cannot be modified, nor do you enable your users to interact with the component. But that's going to change in the next step. You'll provide the component with a local state and permit the deletion of data records. This adjustment starts on the server side.

Extension of the Server Side

On the server side, you need to act in two places: first, you need to make sure that the client receives the initial information, and second, you need to create an interface for

deleting data. In both cases, you access a central data source, which typically is a database. For our simple example, we use a global array with the data. [Listing 18.9](#) shows the extended source code of the server:

```
import { readFileSync } from 'node:fs';
import express from 'express';
import React from 'react';
import ReactDOMServer from 'react-dom/server';
import App from './frontend/src/App';

const port = 3000;
const server = express();

let books = [
  {
    id: '1',
    title: 'JavaScript - The Comprehensive Guide',
    isbn: '978-3836286299',
  },
  {
    id: '2',
    title: 'Clean Code',
    isbn: '978-0132350884',
  },
  {
    id: '3',
    title: 'Design Patterns',
    isbn: '978-0201633610',
  },
];

server.get('/', (request, response) => {
  global.__data__ = books;
  const app = ReactDOMServer.renderToString(<App />);

  const fileContent = readFileSync(
    './frontend/build/index.html',
    'utf-8'
  ).replace(
    '<div id="root"></div>',
    `<div id="root">${app}</div><script>window.__data__ =
      ${JSON.stringify(
        books
      )}</script>`
  );
});
```

```

    response.send(fileContent);
  });

  server.delete('/books/:id', (request, response) => {
    const id = parseInt(request.params.id, 10);
    books = books.filter((book) => book.id !== id);
    response.statusCode = 204;
    response.send();
  });

  server.use(express.static('./frontend/build'));

  server.listen(port, () => {
    console.log(`Server is listening to http://localhost:${port}`);
  });

```

Listing 18.9 Extension of the Server Code (index.js)

For data storage, you first define the `books` structure as an array with the data records you want to display. After that, you make sure that when a client makes a request, not only does the app component get rendered, but a `script` element is included as well that has the `window.__data__` property, which contains the data too.

You also use the `server.delete` method to define an endpoint that allows clients to delete a specific data record.

Extending the Frontend Source Code

In the frontend, the modifications are restricted to the `List` component. There you define a local state and the possibility to delete data. The corresponding source code is shown in [Listing 18.10](#):

```

import { useState } from 'react';

function List() {
  let initialValue = [];
  try {
    initialValue = window.__data__;
  } catch {}

  if (global.__data__) {
    initialValue = global.__data__;
  }

  const [books, setBooks] = useState(initialValue);

```

```
async function handleDelete(id) {
  const response = await fetch(`http://localhost:3000/books/${id}`, {
    method: 'DELETE',
  });
  if (response.ok) {
    setBooks((prevBooks) => prevBooks.filter((book) => book.id !== id));
  }
}
return (
  <table>
    <thead>
      <tr>
        <th>Title</th>
        <th>ISBN</th>
        <th></th>
      </tr>
    </thead>
    <tbody>
      {books.map((book) => (
        <tr key={book.id}>
          <td>{book.title}</td>
          <td>{book.isbn}</td>
          <td>
            <button onClick={() =>
              handleDelete(book.id)}>delete</button>
          </td>
        </tr>
      ))}
    </tbody>
  </table>
);
}
```

```
export default List;
```

Listing 18.10 Adjusting the “List” Component (frontend/src/List.js)

You initialize the state of the `List` component with the data from the server. Here you need to distinguish two cases: Are you working on the server or on the client? The global `window` object doesn't exist on the server. Accessing it causes an exception. For this reason, you should try to set the `initialValue` variable with both `window.__data__` and `global.__data__`, covering both sides. With this state you render the list, to which you add a button per line to delete the respective data record. The deletion process is taken care of by the `handleDelete` function, to which you pass the ID of the data record

to be deleted. The function performs a `DELETE` request for the server and updates the local state of the component so that the record is deleted both on the client side and on the server side.

If you rebuild your frontend with these adjustments and restart the server process, you'll see that the application is rendered on the server side, the client takes control, and you can delete data records. To restore the deleted data, you just need to restart the server process.

When it comes to SSR, React again gives you quite a lot of leeway with regard to custom implementations. But this also means that, in the worst case, you have to reinvent the wheel for every application. However, as SSR is a standard problem, there are already established solutions available in the React ecosystem that you can use. One of the most popular solutions is the Next.js framework, which we'll take a brief look at in the following sections.

18.3 Server-Side Rendering Using Next.js

Next.js is a framework based on React that provides numerous tools and solutions to problems. One of these is a standard setup for SSR.

18.3.1 Initializing a Next.js Application

The goal of Next.js is to take as much work off your hands as possible, so that if possible you only have to bother with your business logic. For this reason, even setting up an application is easy. You use the `npx create-next-app --use-npm --typescript` command to start the interactive setup process, at the end of which you'll have a fully functional Next.js application. During the setup of your application, the setup wizard will ask what name you want to give your application. The answer determines the name of the root directory. Because you want the application here to be a books list again, choose the name `library`. The remaining process runs automatically. It creates an initial file and directory structure for you and downloads all the required dependencies. After successfully initializing your application, you go to the `library` directory and run the `npm install lowdb` command. *LowDB* is a lightweight database that stores your data in a JSON file. You use this database to persist the data.

18.3.2 Implementing the Page Component

Next.js organizes the views of the application into *pages*. These are ordinary React components that you place in the *pages* directory of your application. The special feature of Next.js at this point is that you can address the page components directly via a URL path. For example, if you create a `list.tsx` component, you can access it at `http://localhost:3000/list`. But for demonstration purposes, the default page found in the

index.tsx file in the *pages* directory is sufficient. You delete the contents of this file and replace it with the source code from [Listing 18.11](#):

```
import { GetServerSideProps, NextPage } from 'next';
import { useState } from 'react';
import db from '../db';

type Book = {
  id: string;
  title: string;
  isbn: string;
};

const List: NextPage<{
  initialBooks: Book[];
}> = ({ initialBooks }) => {
  const [books, setBooks] = useState(initialBooks);

  async function handleDelete(id: string) {
    const response = await fetch(`http://localhost:3000/api/books/${id}`, {
      method: 'DELETE',
    });
    if (response.ok) {
      setBooks((prevBooks) => prevBooks.filter((book) => book.id !== id));
    }
  }

  return (
    <table>
      <thead>
        <tr>
          <th>Title</th>
          <th>ISBN</th>
          <th></th>
        </tr>
      </thead>
      <tbody>
        {books.map((book) => (
          <tr key={book.id}>
            <td>{book.title}</td>
            <td>{book.isbn}</td>
            <td>
              <button onClick={() => handleDelete(book.id)}>delete</button>
            </td>
          </tr>
        ))}
      </tbody>
    </table>
  );
};
```

```

    ))}
  </tbody>
</table>
);
};

export const getServerSideProps: GetServerSideProps = async () => {
  await db.read();
  const initialBooks = db.data?.books;

  return {
    props: {
      initialBooks,
    },
  };
};

export default List;

```

Listing 18.11 “List” Component in Next.js (pages/index.tsx)

This component already contains the entire functionality from the previous example. The component itself has the `NextPage` type and receives the initial value for the state in which you keep the book list through the SSR. The `handleDelete` function communicates with the server via the `/api/books/` path and appends the ID of the record to be deleted. Once the deletion has been successful, you update the state and React will update the component for the users. The JSX structure of the component consists of a table element where you display the titles and ISBNs of the books. You’ll also render one button each to delete the respective data record.

What’s specific to Next.js is the `getServerSideProps` function, which ensures that the component is prerendered on the server side. In this function, you use the `read` method of `LowDB` to read the data source, and then you can access the data via `db.data.books`. The object structure returned by this function makes sure that the information is available to you as props in the component.

18.3.3 Implementing the Server Side

For storing data, you use the `lowdb` package and a JSON file. You save this JSON file as `data.json` in the root directory of your application. The content is shown in [Listing 18.12](#):

```

{
  "books": [
    {
      "id": "1",

```

```
    "title": "JavaScript - The Comprehensive Guide",
    "isbn": "978-3836286299"
  },
  {
    "id": "2",
    "title": "Clean Code",
    "isbn": "978-0132350884"
  },
  {
    "id": "3",
    "title": "Design Patterns",
    "isbn": "978-0201633610"
  }
]
```

Listing 18.12 Data Source for the Next.js Application (data.json)

You initialize LowDB centrally in a file named *db.ts*, because you need it in several places in your application—for example, in the `getServerSideProps` function of the `List` component. [Listing 18.13](#) contains the source code of this file:

```
import { join, dirname } from 'node:path';
import { fileURLToPath } from 'node:url';
import { Low, JSONFile } from 'lowdb';

const __dirname = dirname(fileURLToPath(import.meta.url));

type Data = {
  books: Array<{
    id: string;
    title: string;
    isbn: string;
  }>;
};

const file = join(__dirname, 'data.json');
const adapter = new JSONFile<Data>(file);
const db = new Low(adapter);

export default db;
```

Listing 18.13 Initialization of the Database (db.ts)

Using the `JSONFile` class, you create an adapter for LowDB that ensures that a file is used as a storage medium. In conjunction with the adapter you just created, the `Low`

class will create the database object for you to manage your data. To allow you to work type-safe in your application, `lowdb` allows you to pass the structure of the database to the `JSONFile` class, which is implemented as a generic class.

With this state of the source code, you can start your application on the command line using the `npm run dev` command in development mode or via `npm run build` and `npm start` in production mode. Your browser will then display the book list. When you look at the developer tools of your browser, you can see that the component is already pre-rendered on the server side. What doesn't work yet is the deletion of data records.

18.3.4 API Routes in Next.js

On the server side, Next.js works in a Node.js process to provide SSR and generates source code that you can run in the browser, and the framework enables you to define so-called API routes. These are endpoints that, for example, deliver data or allow write access to the database. Next.js provides you with a similar mechanism here as it did for the page components. Below the `pages` directory, there is a directory named `api`. There you can create files and directories that will serve as your endpoints. For deleting data records, you use the `/api/books/:id` path, where `:id` stands for the ID of the record to be deleted. You cover these variables with a special file name. For example, you create a file named `[id].ts` in the `pages/api/books` directory. The source code of this file is shown in [Listing 18.14](#):

```
import type { NextApiRequest, NextApiResponse } from 'next';
import db from '../../db';

export default async function handler(
  request: NextApiRequest,
  response: NextApiResponse
) {
  if (request.method === 'DELETE') {
    await db.read();
    const { id } = request.query;

    if (db.data) {
      db.data.books = db.data?.books.filter((book) => book.id !== id);
    }
    db.write();

    response.statusCode = 204;
    response.end();
  }
}
```

Listing 18.14 Implementation of a Dynamic Route (`pages/api/books/[id].ts`)

An API route file exports a handler function, which receives a `request` object and a `response` object. Based on the incoming request, you can decide what needs to be done. In the example case, you simply respond to the `DELETE` method, connect to the database, extract the ID from the URL path via the `request.query` property, and then delete the data record from the database. In response, you send status code 204 to the client to indicate that the operation was successful.

This adjustment allows you to view the list rendered on the server side and dynamically delete data without the need to reload the page. In [Figure 18.3](#), you can see the source code that Next.js delivers when you ask for the list.

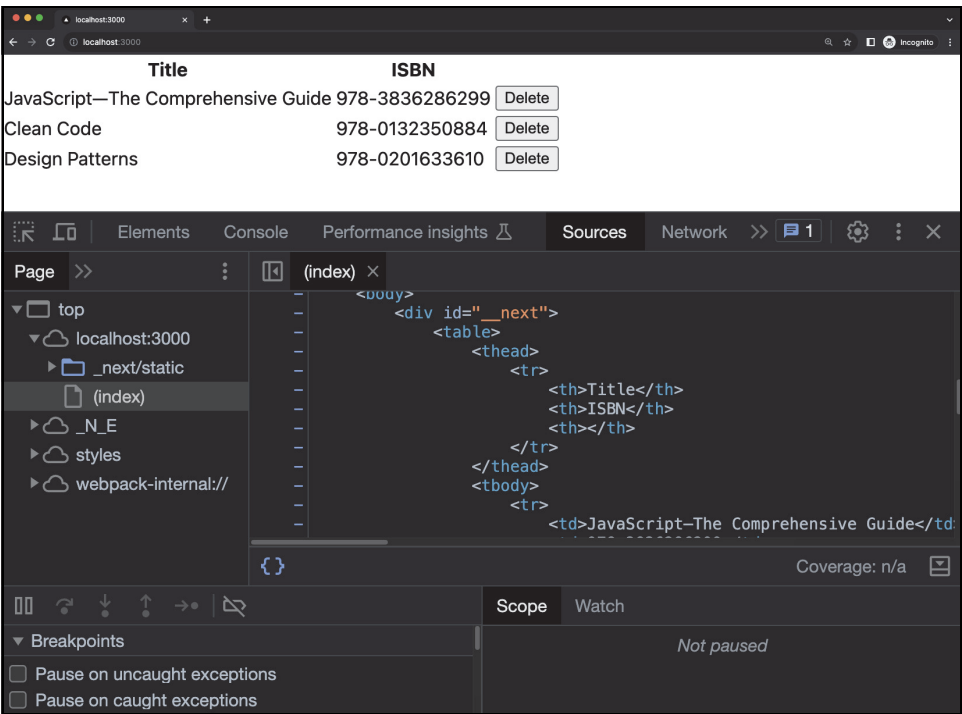


Figure 18.3 SSR in Next.js

In this example, you’ve seen how Next.js can help you with standard tasks like SSR and that there is no need for you to configure Webpack and Babel yourself.

18.4 Summary

This chapter has provided an overview of server-side rendering:

- React itself provides the required tools for SSR but doesn’t have a prebuilt implementation. That’s why you need to resort either to your own implementation or to a third-party library such as Next.js.

- Using a combination of Node.js, Webpack, and Babel, you can implement SSR for your application yourself.
- In SSR, the client requests the application from the server, but instead of delivering the individual resources, the server prepares the HTML structure so that it can be displayed directly by the browser. The `renderToString` function of React is used for this purpose.
- In addition to the static HTML structure, you can deliver dynamic data in the form of global JavaScript variables. These are used to prevent requests to the server to initialize components.
- Once the prerendered data arrives in the browser, React takes control by running the `hydrateRoot` function.
- SSR allows the application to be displayed to the user more quickly. This doesn't necessarily mean that a user can also interact faster with the application. The hydrate process must first be successfully completed for this.
- In the React ecosystem, there are frameworks like Next.js that allow you to perform SSR without any additional configuration.
- To enable SSR in Next.js, you implement the `getServerSideProps` function in a page component. The framework then takes care of everything else.

In the following chapter, you'll learn about the features provided by React to optimize the performance of your application.

Chapter 19

Performance

React is basically a performant solution for single-page applications. However, the developers of the library are focusing their development efforts on making React even faster and more usable. For this reason, there are some features you should know about to make your application even faster.

When performance tuning an application, it's true that you don't need to eke out the very last bit of performance in every case. In many cases, optimizing the source code affects its readability. So you should make sure that the readability and consistency of your source code are at the forefront of your development. Only when you notice that the performance of your application isn't satisfactory should you start to really optimize your source code with regard to performance. But please don't proceed blindly either. First carry out a measurement of the workflow you want to improve. Then make the improvement to your source code and measure again afterward. If the change to the code has led to noticeably improved performance, the optimization was successful. If that's not the case, you should undo the change.

If you implement your application's interface using React, the library updates the display whenever the data is changed. For this purpose, React has some optimizations. For example, only the parts of the component tree that have changed will be redrawn. What exactly should be displayed is determined by the JSX structure returned by the component. For this purpose, the component function or the `render` method will be executed with each prop and state change. Both the creation of function objects and the check for whether a component has to be drawn have an impact on the performance of an application.

19.1 The Callback Hook

One pattern you'll find relatively often in a React application that can have a negative impact on performance is the definition of functions within the component function. It may sound exotic at first, but you use this, for example, whenever you write a callback function for a `click` event, as in [Listing 19.1](#):

```
import React, { useState } from 'react';

type Props = {
  time: number;
```

```
};

const Counter: React.FC<Props> = ({ time }) => {
  const [count, setCount] = useState(0);

  return (
    <div>
      <div>{count}</div>
      <div>
        <button
          onClick={() => {
            setCount(count + 1);
          }}
        >
          increment
        </button>
      </div>
      <div>{time}</div>
    </div>
  );
};

export default Counter;
```

Listing 19.1 Click Handler in a Function Component (src/Counter.tsx)

This component uses the state hook to create a local state as a number and display it. A button enables you to increase this number by the value 1. The component receives a numeric time value via its props, which represents the current time and which is increased by the value 1000 every second. The corresponding *App.tsx* component that integrates the `Counter` component is shown in [Listing 19.2](#):

```
import React, { useState, useEffect } from 'react';
import Counter from './Counter';

const App: React.FC = () => {
  const [time, setTime] = useState(Date.now());

  useEffect(() => {
    const interval = setInterval(() => {
      setTime(time + 1000);
    }, 1000);
    return () => clearInterval(interval);
  }, [time]);
```

```

    return <Counter time={time} />;
  };

  export default App;

```

Listing 19.2 Integration of the “Counter” Component (src/App.tsx)

Every second, the prop values of the `Counter` component change, so it must be redrawn. However, this also means that you have to recreate the click handler each time you render. The callback hook is a solution in this context that allows you to prevent a new function object from being created with every render operation. [Listing 19.3](#) shows the use of the callback hook, using the `Counter` component as an example:

```

import React, { useCallback, useState } from 'react';

type Props = {
  time: number;
};

const Counter: React.FC<Props> = ({ time }) => {
  const [count, setCount] = useState(0);

  const handleClick = useCallback(() => {
    setCount((prevCount) => prevCount + 1);
  }, []);

  return (
    <div>
      <div>{count}</div>
      <div>
        <button onClick={handleClick}>increment</button>
      </div>
      <div>{time}</div>
    </div>
  );
};

export default Counter;

```

Listing 19.3 Using the Callback Hook (src/Counter.tsx)

By means of the callback hook, React regenerates the callback function only when the dependencies change that you specify in the second argument. If you don’t want the function to change, you specify an empty array as in the example. Then React creates the callback function only once and reuses it between each render operation. If you use

an external variable in the callback function, React warns you and recommends that you pass this variable as a dependency.

A similar functionality is hidden behind the memo hook. However, this hook returns a calculated value instead of a callback function. The memo hook is mainly used for complex calculations. You pass a function to the `useMemo` function that calculates the value. As the second argument, you again pass an array of dependencies for which you want the value to be recalculated when they change.

19.2 Pure Components

Pure components represent a further performance optimization. They are class components that derive from `React.PureComponent` instead of `React.Component`. The difference between a regular and a pure component is that the `PureComponent` class implements the `shouldComponentUpdate` method with a simple comparison of the `state` and `props` properties. In this simplified comparison lies the potential performance advantage of pure components.

The prerequisite, however, is that the data types of the `props` and `state` are simple values and not objects or arrays that are passed by reference and can lead to incorrect results. As an example, in [Listing 19.4](#) you can see a `Book` component that changes its state every second:

```
import { PureComponent } from 'react';

class Book extends PureComponent<unknown, { title: string }> {
  constructor(props: unknown) {
    super(props);
    this.state = {
      title: 'Design Patterns',
    };
  }

  componentDidMount() {
    setInterval(() => {
      console.log('interval');
      this.setState({ title: 'Design Patterns' });
    }, 1000);
  }

  render() {
    console.log('render');
    return <h1>{this.state.title}</h1>;
  }
}
```

```
}
```

```
export default Book;
```

Listing 19.4 Implementation of “PureComponent” (src/Book.tsx)

This component has its own `title` state, which you initialize to the `Design Patterns` value. In the `componentDidMount` method, you start an interval that sets the state every second, but always to the same value. In the `render` method, you output the title in an `h1` element.

To make the effect of `PureComponent` more visible, you write `interval` to the console inside the `setInterval` callback function and the `render` string in the `render` method. If you include this component in your `App` component, you’ll see the `interval` string every second, whereas you’ll see the `render` string only once. The reason is that React prevents the component from being rendered again with the same state via the `PureComponent`. If you then replace the `PureComponent` with `Component`, any state change will cause the component to be rendered again, regardless of whether the value is the same.

A similar implementation exists in the form of the `React.memo` function.

19.3 “React.memo”

`React.memo` is a higher-order component, which doesn’t re-render the passed component until the props change. Again, a simple comparison is used. To demonstrate this optimization, we first take a look at a nonoptimized version of two components in [Listing 19.5](#):

```
import React, { useEffect, useState } from 'react';

type Props = {
  title: string;
};

const Headline: React.FC<Props> = ({ title }) => {
  console.log('render headline');
  return <h1>{title}</h1>;
};

const App: React.FC = () => {
  const [state, setState] = useState({ number: 0, title: 'Hello Memo' });

  useEffect(() => {
    setTimeout(() => {
      setState((prevState) => ({ ...prevState, number: 1 }));
    });
  });
};
```



```
    }, 1000);
    setTimeout(() => {
      setState((prevState) => ({
        ...prevState,
        number: 2,
        title: 'Hello React',
      }));
    }, 2000);
  }, []);
  return <Headline title={state.title} />;
};
```

```
export default App;
```

Listing 19.5 Component without “React.memo” (src/App.tsx)

Contrary to the usual convention, the file contains two components to keep the example compact. The parent component—App—has its own state with the `number` and `title` properties. In the effect hook, you change the `number` property after one second and the `number` and `title` properties after two seconds.

Without optimization, this results in the child component `Headline` being rendered a total of three times: the first time it is initially rendered; the second time the component is re-rendered because the state of the parent component changes; and the third time the state of the parent component changes again, but so do the props of the component itself. This means that the second rendering of the component is actually unnecessary. And that's where `React.memo` comes in: if you wrap the `Headline` component in the `memo` function, as you can see in [Listing 19.6](#), the component will only be rendered twice.

```
import React, { ReactElement, useEffect, useState, memo } from 'react';

type Props = {
  title: string;
};

const Headline = memo(function ({ title }: Props): ReactElement {
  console.log('render headline');
  return <h1>{title}</h1>;
});

const App: React.FC = () => {
  const [state, setState] = useState({ number: 0, title: 'Hello Memo' });

  useEffect(() => {...}, 2000);
  }, []);
```

```

    return <Headline title={state.title} />;
  };

export default App;

```

Listing 19.6 Use of “React.memo” (src/App.tsx)

Using the `memo` function, every time you change the props, React checks to see if they have changed by means of a simple comparison—which isn’t the case with the second render operation. In this case, React doesn’t re-render the component, but uses the previous version of the component.

In the example, the `Headline` component is a simple stateless component. If a component has its own state or uses the Context API, React renders the component again in any case if one of the two structures changes. The `memo` function works only with the props of the component.

If your component's props are objects and you’re only interested in certain properties, the default `memo` variant will no longer work. In this case, you can pass a function as the second argument to the function in order to determine when exactly you consider the props structure to be the same. In [Listing 19.7](#), you can see how this works in the `Headline` component:

```

import React, { ReactElement, useEffect, useState, memo } from 'react';

type Props = {
  data: {
    number: number;
    title: string;
  };
};

function isEqual(prevProps: Props, nextProps: Props) {
  return prevProps.data.title === nextProps.data.title;
}

const Headline = memo(function ({ data: { title } }: Props): ReactElement {
  console.log('render headline');
  return <h1>{title}</h1>;
}, isEqual);

const App: React.FC = () => {
  const [state, setState] = useState({ number: 0, title: 'Hello Memo' });

  useEffect(() => {...}, []);

```

```
    return <Headline data={state} />;  
  };  
  
export default App;
```

Listing 19.7 Custom Comparison of the Props Structures (src/App.tsx)

You can simulate the `memo` issue in the context of an object structure by passing the entire `State` object to the component instead of the `title` property. Without any further change, React will render the component another three times. But if you then define an `isEqual` function, which receives the current and future versions of the props as arguments, you can decide there whether or not the component should be rendered. If you return `true`, React won't render the component again. If `false`, React will render the component again. With this adjustment, only two render operations of the `Headline` component take place again.

In addition to reducing unnecessary render operations, React provides `Suspense` for other optimizations in your application.

19.4 “React.lazy”: “Suspense” for Code Splitting

One of the big problems with SPAs is the *bundle size*, which is the size of the package you send from the server to the client so that it can render the application. Build tools like Webpack already have some extensions that you can use to significantly reduce the bundle size. These extensions mainly include so-called manglers and compressors such as Terser. These tools remove unnecessary whitespaces and comments from your source code and shorten the names of variables. Another optimization is so-called tree-shaking, in which Webpack analyzes your source code and removes unused parts.

In addition, *lazy loading* allows you to selectively cut components from your initial bundle and reload them at a later time. The core of this feature is the `lazy` function of React.

19.4.1 Lazy Loading in an Application

To asynchronously reload a component or an entire component tree, which is nothing more than lazy loading, you simply replace the component's `import` statement. If you then conditionally render the component, React reloads the required files. To give your users feedback that something is happening in the background during this loading time, you use the `Suspense` component of React and then have the option to display fallback content.

Lazy loading allows you to build your application's bundle to include only the components your application needs for initial display. You then reload all other components

as required. As an example of lazy loading, we can implement a list of books with the option to view the details of a data record. The `List` component, which is responsible for the list display, is part of the initial bundle. You reload the detail view asynchronously using `React.lazy`.

The `Details` component is independent of lazy loading. In [Listing 19.8](#), you can see the source code of this component.

```
import React from 'react';
import { Book } from './Book';

type Props = {
  book: Book;
};

const Details: React.FC<Props> = ({ book }) => {
  return (
    <table>
      <tbody>
        <tr>
          <th>Title: </th>
          <td>{book.title}</td>
        </tr>
        <tr>
          <th>Author: </th>
          <td>{book.author}</td>
        </tr>
        <tr>
          <th>ISBN: </th>
          <td>{book.isbn}</td>
        </tr>
        <tr>
          <th>Rating: </th>
          <td>{book.rating}</td>
        </tr>
      </tbody>
    </table>
  );
};

export default Details;
```

Listing 19.8 Implementation of the “Details” Component (`src/Details.tsx`)

The `Details` component receives the data to display via its props and then displays the individual properties in a table. The `List` component has significantly more to do with lazy loading. [Listing 19.9](#) contains its source code.

```
import React, { lazy, Suspense, useState } from 'react';
const Details = lazy(() => import('./Details'));
const books = [
  {
    id: 1,
    title: 'JavaScript - The Comprehensive Guide',
    author: 'Philip Ackermann',
    isbn: '978-3836286299',
    rating: 5,
  },
  {
    id: 2,
    title: 'Clean Code',
    author: 'Robert Martin',
    isbn: '978-0132350884',
    rating: 2
  },
  {
    id: 3
    title: 'Design Patterns',
    author: 'Erich Gamma',
    isbn: '978-0201633610',
    rating: 5,
  },
];
const List: React.FC = () => {
  const [details, setDetails] = useState<number | null>(null);
  return (
    <>
      {books.map((book) => (
        <div key={book.id}>
          {book.title}
          <button onClick={() => setDetails(book.id)}>Details</button>
          <Suspense fallback={<div>...loading</div>}>
            {details === book.id && <details book={book} />}
          </Suspense>
        </div>
      )))}
    </>
  );
};
```

```
};
```

```
export default List;
```

Listing 19.9 Implementation of the “List” Component with Lazy Loading (src/List.tsx)

The `List` component has a local state that you use to control for which data record you want to display the details. In the JSX structure, you render a `div` element for each data record, in which you display the title and a button. When you click the button, you set the `details` state in the background, which in turn causes the condition to display the details for the record to become true. React then loads the source code of the `Details` component from the server. In the meantime, you display the `...loading` string. If the source code of the component is available, React displays the component with the desired data record. When you click another button, the component is already loaded and there is no further delay.

To enable lazy loading, you replace the import of a component with the call of the `lazy` component with the dynamic `import` statement and encapsulate the representation of the component in the `Suspense` component.

As an alternative to the `div` element with the string, you can render any component; you are not limited to simple elements. Typical examples include loading indicators, which you don’t need to implement yourself. You can also use prebuilt packages such as `react-spinners`.

The final step you need to take before you can test lazy loading is to integrate it with your `App` component. In [Listing 19.10](#), you can see how this works:

```
import React from 'react';
import List from './List';

const App: React.FC = () => {
  return <List />;
}

export default App;
```

Listing 19.10 Integration of the “List” Component in the Application (src/App.tsx)

Now when you open your application in the browser and click on one of the **Details** buttons, you’ll see an additional file loaded in development mode in the **Network** tab of your developer tools: `src_Details_tsx.chunk.js`. [Figure 19.1](#) shows what this looks like in the browser.

Lazy loading isn’t limited to conditional rendering; you can also use it in conjunction with `React Router`.

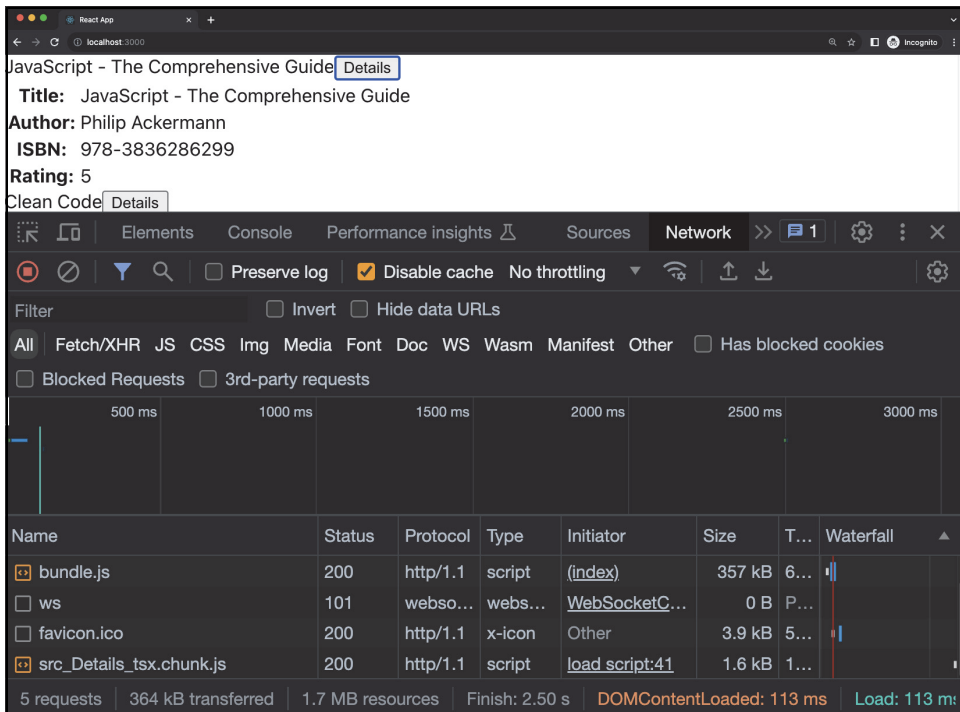


Figure 19.1 Lazy Loading in the Developer Tools

19.4.2 Lazy Loading with React Router

Basically, you can use the components you load via lazy loading anywhere in your application. If you use React Router in your application, you'll have predetermined breaking points in the component tree of your application anyway with the different routes. It's exactly at these points that you can also integrate lazy loading. As an example, we'll again use the list and detail views here, with the difference that you can now access both via different routes. The basis for this example is the `List` component in the `List.tsx` file and the `Details` component in `Details.tsx` (both in the `src` directory), which you can find in [Listing 19.11](#) and [Listing 19.12](#):

```
import React from 'react';
import { Link } from 'react-router-dom';
import { Book } from './Book';

type Props = {
  books: Book[];
};

const List: React.FC<Props> = ({ books }) => {
  return (
```

```

    <>
      {books.map((book) => (
        <div key={book.id}>
          {book.title}
          <Link to={`/details/${book.id}`}>Details</Link>
        </div>
      ))}
    </>
  );
};

```

```
export default List;
```

Listing 19.11 Implementation of the “List” Component for Lazy Loading with React Router (src/List.tsx)

```

import React from 'react';
import { Book } from './Book';
import { useParams } from 'react-router-dom';

type Props = {
  books: Book[];
};

const Details: React.FC<Props> = ({ book }) => {
  const id = parseInt(useParams<{ id: string }>().id!, 10);
  const book = books.find((b) => b.id === id);

  if (!book) {
    return <div>No data found</div>;
  }

  return (
    <table>
      <tbody>
        <tr>
          <th>Title: </th>
          <td>{book.title}</td>
        </tr>
        <tr>
          <th>Author: </th>
          <td>{book.author}</td>
        </tr>
        <tr>
          <th>ISBN: </th>

```



```
        <td>{book.isbn}</td>
      </tr>
      <tr>
        <th>Rating: </th>
        <td>{book.rating}</td>
      </tr>
    </tbody>
  </table>
);
};
```

```
export default Details;
```

Listing 19.12 Implementation of the “Details” Component for Lazy Loading with React Router (src/Details.tsx)

Both components are implemented independently of lazy loading and receive their data via their props for simplicity. The source code of the App component is shown in [Listing 19.13](#). This component is responsible for static data management as well as for the routing configuration and lazy loading.

```
import React, { lazy, Suspense } from 'react';
import { BrowserRouter, Route, Routes, Navigate } from 'react-router-dom';
const Details = lazy(() => import('./Details'));
const List = lazy(() => import('./List'));

const books = [
  {
    id: 1,
    title: 'JavaScript - The Comprehensive Guide',
    author: 'Philip Ackermann',
    isbn: '978-3836286299',
    rating: 5,
  },
  {
    id: 2,
    title: 'Clean Code',
    author: 'Robert Martin',
    isbn: '978-0132350884',
    rating: 2
  },
  {
    id: 3
    title: 'Design Patterns',
    author: 'Erich Gamma',
```

```

    isbn: '978-0201633610',
    rating: 5,
  },
];

const App: React.FC = () => {
  return (
    <BrowserRouter>
      <Suspense fallback={<div>...loading</div>}>
        <Routes>
          <Route path="/list" element={<List books={books} />} />
          <Route path="/details/:id" element={<Details books={books} />} />
          <Route path="/" element={<Navigate to="/list" />} />
        </Routes>
      </Suspense>
    </BrowserRouter>
  );
};

export default App;

```

Listing 19.13 Lazy Loading with React Router (src/App.tsx)

So that you don't have to worry about loading the data from the server, the information for display is contained in the static `books` array. Both components are loaded with `React.lazy` and the `books` array is passed to both components via the `books` prop. In the JSX structure of the `App` component, you first integrate `BrowserRouter` and then make the `Suspense` component display the `...loading` text while the desired components aren't yet available. In the route definition itself, you reference the `List` and `Details` components. React then makes sure that the components are loaded as soon as the respective route is activated.

With lazy loading, you've now become acquainted with one use case for the `Suspense` components. Another area of use is the loading of data.

19.5 Suspense for Data Fetching

As you've already seen with lazy loading, you can use the `Suspense` component to inform your users that an asynchronous operation is running in the background. However, you can use `Suspense` not only in combination with `React.lazy`, but also for loading data.

In the simplest case, you can think of `Suspense` for Data Fetching as a built-in load state that you display while the data is being loaded.

The problem with Suspense for Data Fetching is that the `Suspense` component doesn't know about the data loading. For this purpose, you need a separate interface, but it must be compatible with `Suspense`. This means the `Fetch` API of the browser cannot be used in this case. The initial implementation of Suspense for Data Fetching was implemented by React developers using *Relay*, a GraphQL client from Facebook that supported this feature at a very early stage of `Suspense`.

However, there are more libraries that support `Suspense`, such as `React Query`. You can use this library primarily for loading data from a server interface. In addition, `React Query` enables you to cache data and also send data to the server and it supports the `Suspense` component of React. But before we get into that, you'll learn how to use `React Query` in your application in general.

19.5.1 Installing and Using React Query

React Query is a library that takes care of loading data for you, managing the state of a query, and caching queries. You can install `React Query` using the `npm install @tanstack/react-query` command. You don't need to install additional type definitions for use with `TypeScript` because the library itself is written in `TypeScript` and comes with its type definitions.

In preparation for the following example, you install the `json-server` package in your application via the `npm install json-server` command. You also define a file named `data.json` in the root directory of your application. The contents of this file are shown in [Listing 19.14](#):

```
{
  "books": [
    {
      "id": 1,
      "title": "JavaScript - The Comprehensive Guide",
      "author": "Philip Ackermann",
      "isbn": "978-3836286299",
      "rating": 5
    },
    {
      "id": 2,
      "title": "Clean Code",
      "author": "Robert Martin",
      "isbn": "978-0132350884",
      "rating": 4
    },
    {
      "id": 3
      "title": "Design Patterns",
      "author": "Erich Gamma",
```

```

    "isbn": "978-0201633610",
    "rating": 5
  }
]
}

```

Listing 19.14 Data Basis for the React Query Example (data.json)

To be able to start the backend without any problem, you add another entry to your *package.json* file in the *scripts* section. You can see this in [Listing 19.15](#):

```

{
  ...
  "scripts": {
    ...
    "backend": "json-server -w data.json -p 3001"
  },
  ...
}

```

Listing 19.15 Start Script for the Backend (package.json)

With this source code, you can start the backend of your application via the `npm run backend` command. The next step is to integrate React Query into your application. Like many other libraries, this library uses the React context, so you need to include the provider in a central place. In the example, this is done in the *App* component, as you can see in [Listing 19.16](#):

```

import React from 'react';
import { QueryClient, QueryClientProvider } from 'react-query';

import List from './List';

const queryClient = new QueryClient();

const App: React.FC = () => {
  return (
    <QueryClientProvider client={queryClient}>
      <List />
    </QueryClientProvider>
  );
};

export default App;

```

Listing 19.16 Integration of “QueryClientProvider” into the Application (src/App.tsx)

With this preliminary work done, you can now make sure in the `List` component that the data gets loaded from the server and displayed:

```
import React from 'react';
import { useQuery } from 'react-query';
import { Book } from './Book';

async function getBooks(): Promise<Book[]> {
  const response = await fetch('http://localhost:3001/books');
  if (!response.ok) {
    throw new Error('Response not OK');
  }
  const data = await response.json();
  return data;
}

const List: React.FC = () => {
  const { data, isLoading, isError } = useQuery(['books'], getBooks);

  if (isLoading) {
    return <div>...loading data</div>;
  }

  if (isError) {
    return <div>{`An error has occurred`}</div>;
  }

  return (
    <ul>
      {data?.map((book) => (
        <li key={book.id}>{book.title}</li>
      ))}
    </ul>
  );
};

export default List;
```

Listing 19.17 Implementation of the “List” Component with React Query (src/List.tsx)

The implementation of the `List` component in [Listing 19.17](#) is divided into two parts: first you implement the `getData` function to load the data and then the actual `List` component. API functions like `getData` should be moved to a separate file, at least if you need them in multiple places in your application. To keep the example compact, we implement them in one file. This `getBooks` function simply encapsulates the `fetch`

request and returns a `Promise` that resolves with the data if successful and rejects it if it fails.

You use this function in the component together with the `useQuery` hook of React Query to load the data from the server. First you pass the query key to `useQuery`. The *query key* is an array that React Query can use to uniquely identify the query.

The second argument is the fetcher function that you use to load the data from the server. React Query takes care of data management in the `useQuery` function and gives you access to various information via the returned object. Thus you can use the `isLoading` property to find whether the server request is currently being processed. The `isError` property has the value `true` if an error occurred. Finally, in the `data` property you'll find the data you loaded from the server. This case shows the simplest way to integrate React Query, with read-only access.

Deleting Data Records Using React Query Mutations

If you now extend your component so that your users can also delete data, you must know that you invalidate the cache of React Query:

```
import React from 'react';
import { useMutation, useQuery, useQueryClient } from 'react-query';
import { Book } from './Book';

async function getBooks(): Promise<Book[]> {...}

async function removeBook(id: number) {
  const response = await fetch(`http://localhost:3001/books/${id}`, {
    method: 'DELETE',
  });
  if (!response.ok) {
    throw new Error('Response not OK');
  }
}

const List: React.FC = () => {
  const queryClient = useQueryClient();

  const { data, isLoading, isError } = useQuery(['books'], getBooks);
  const mutation = useMutation(removeBook, {
    onSuccess() {
      queryClient.invalidateQueries(['books']);
    },
  });
  if (isLoading) {
    return <div>...loading data</div>;
  }
}
```

```
if (isError) {
  return <div>{`An error has occurred`}</div>;
}

return (
  <ul>
    {data?.map((book) => (
      <li key={book.id}>
        {book.title}
        <button
          onClick={() => {
            mutation.mutate(book.id);
          }}
        >
          delete
        </button>
      </li>
    ))}
  </ul>
);
};

export default List;
```

Listing 19.18 Deletion of Data Records (src/List.tsx)

Similar to the previous example for reading the data, when you delete a record, you first define a function that encapsulates the server communication for you. Then, within the component, you use the `useQueryClient` function to implement a reference to the React Query client, which you can use later to specifically invalidate the cache.

You use the `useMutation` function to create a mutation object via the `removeBook` API function and a configuration object. For example, in this object you can define an `onSuccess` method that executes React Query if the server request was successful. Here you use the `invalidateQueries` method to invalidate the query by means of the `['books']` query key. React Query then reloads the data and updates the display.

You can now trigger the mutation via `button` elements in the list by calling the `mutate` method with the ID of the record to be deleted.

19.5.2 React Query and Suspense

React Query supports Suspense for Data Fetching (as do many other libraries, like `relay` or `SWR`). If you use this feature, state management is even more deeply integrated into React and you don't need to worry about displaying the loading state and error

handling yourself. For Suspense to work with React Query, you must enable Suspense mode. You can achieve this either globally when creating `QueryClient` or, as in our example, locally per query.

For the following example, you install the `react-error-boundary` package using the `npm install react-error-boundary` command. It provides `ErrorBoundary` so that you don't have to implement it yourself. In the first step, you wrap the `List` component into `ErrorBoundary` to handle the errors correctly and into the `Suspense` component to cover the load state. The corresponding source code of the `App` component for the example is shown in [Listing 19.19](#):

```
import React, { Suspense } from 'react';
import { QueryClient, QueryClientProvider } from 'react-query';
import { ErrorBoundary } from 'react-error-boundary';

import List from './List';

const queryClient = new QueryClient();

const App: React.FC = () => {
  return (
    <QueryClientProvider client={queryClient}>
      <ErrorBoundary
        FallbackComponent={({ error }) => <div>{error.message}</div>}
      >
        <Suspense fallback={<div>...loading data</div>}>
          <List />
        </Suspense>
      </ErrorBoundary>
    </QueryClientProvider>
  );
};

export default App;
```

Listing 19.19 Integration of the “ErrorBoundary” and “Suspense” Components (`src/App.tsx`)

You can pass a component to the `ErrorBoundary` component via the `FallbackComponent` prop, which receives the representation of the occurred error as a prop. The `Suspense` component behaves similarly to lazy loading, displaying the contents of the `fallback` prop while the data is being loaded. The `List` component is a little easier to use with the addition of `Suspense`, as you can see in [Listing 19.20](#):


```
import React from 'react';
import { useMutation, useQuery, useQueryClient } from 'react-query';
import { Book } from './Book';

async function getBooks(): Promise<Book[]> {...}

async function removeBook(id: number) {...}

const List: React.FC = () => {
  const queryClient = useQueryClient();

  const { data } = useQuery(['books'], getBooks, { suspense: true });

  const mutation = useMutation(removeBook, {
    onSuccess() {
      queryClient.invalidateQueries(['books']);
    },
  });

  return (<ul>...</ul>);
};

export default List;
```

Listing 19.20 Activating “Suspense” Mode (src/List.tsx)

As a third argument, you can pass a configuration object to the `useQuery` function in which you can set the `suspense` property to the value `true`. This property activates the Suspense mode so that `ErrorBoundary` and the `Suspense` component are addressed correctly.

19.5.3 Concurrency Patterns

The capabilities of Suspense for Data Fetching result in several concurrency patterns that you can implement in React. These include the following:

- **Fetch on render**

This mode is the default in React. You render a component that requires data from the server to display. The component loads the data in its effect hook and stores the data in the local state, which results in a re-rendering.

- **Fetch then render**

In the second approach, you load the data in an outer component and don’t render the actual component until the data is available. Then you pass the data to the


```

    </thead>
    <tbody>
      {books.map((book) => (
        <tr key={book.id}>
          <td>{book.title}</td>
          <td>{book.author}</td>
          <td>{book.isbn}</td>
          <td>{book.rating} && <span>{'.'.repeat(book.rating)}</span></td>
        </tr>
      ))}
    </tbody>
  </table>
);
};

export default List;

```

Listing 19.21 Representation of a Very Extensive List (src/List.tsx)

Outside the component, you create an array with 100,000 elements and fill each with an object. Subsequently, each object receives a unique ID. You use this array to render a table in the `List` component and display the information from the array. The browser struggles with these 100,000 entries, but it displays the table. If you increase the number of data records by increasing the number of elements of the array, at some point the browser will tell you that an error occurred while rendering the page. For me, the error occurred with one million entries. You can see the display in the browser in [Figure 19.2](#).

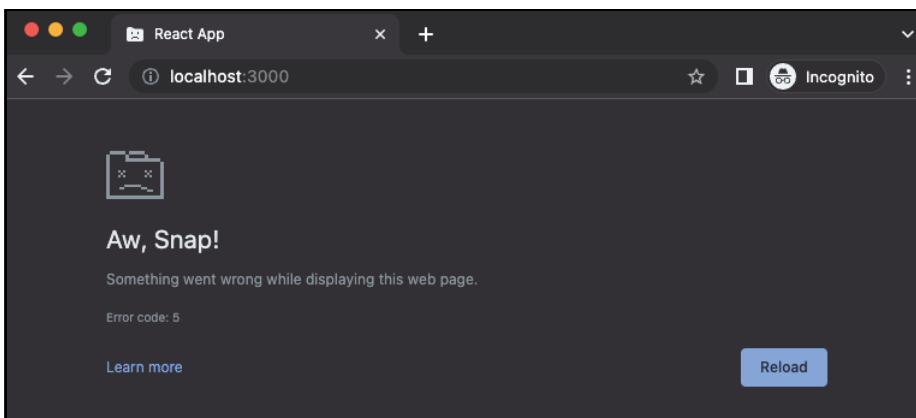


Figure 19.2 Error Message during Display

There are situations in which you have to work with such large amounts of data. A slightly longer waiting time usually is still tolerated, but an error message in the

browser is not. In this case, you have two options. The first is to build pagination into your presentation so that you display only a portion of the data and let your users navigate. The problem here is that the usability of such lists is significantly limited as users have to go through the list page by page and cannot scroll.

An alternative to this is so-called virtual scrolling. You simulate the display of the entire list but show users only the area that is actually visible and render additional records as needed. This is a commonly used solution, so reference implementations and libraries are available and you don't need to worry about implementing them yourself. An example of such a library is `react-window`. You can install the package via the `npm install react-window` command and then use it as shown in [Listing 19.22](#). For TypeScript support, you also need to install the type definitions via `npm install --save-dev @types/react-window`.

```
import React, { CSSProperties, ReactNode } from 'react';

import { FixedSizeList as List, ListChildComponentProps } from 'react-window';

const books = new Array(1_000_000)
  .fill({
    title: 'Design Patterns',
    author: 'Erich Gamma',
    isbn: '978-0201633610',
    rating: 5,
  })
  .map((book, id) => ({ ...book, id }));

type InnerProps = {
  children: ReactNode;
  style: CSSProperties;
};

const Inner: React.FC<InnerProps> = ({ children, style }) => {
  return (
    <table>
      <thead>
        <tr>
          <th style={{ width: 50 }}>ID</th>
          <th style={{ width: 200 }}>Title</th>
          <th style={{ width: 200 }}>Author</th>
          <th style={{ width: 200 }}>ISBN</th>
          <th style={{ width: 200 }}>Rating</th>
        </tr>
      </thead>
      <tbody style={{ ...style, position: 'absolute', width: '100%' }}>
```

```
        {children}
      </tbody>
    </table>
  );
};

const Row: React.FC<ListChildComponentProps> = ({ index, style }) => {
  return (
    <tr style={style}>
      <td style={{ width: 50 }}>{books[index].id}</td>
      <td style={{ width: 200 }}>{books[index].title}</td>
      <td style={{ width: 200 }}>{books[index].author}</td>
      <td style={{ width: 200 }}>{books[index].isbn}</td>
      <td style={{ width: 100 }}>{books[index].rating}</td>
    </tr>
  );
};

const App: React.FC = () => {
  return (
    <List
      height={300}
      itemCount={books.length}
      itemSize={30}
      width={750}
      innerElementType={Inner}
    >
      {Row}
    </List>
  );
};

export default App;
```

Listing 19.22 Implementation of a Virtual Table (src/App.js)

The core of this example is the `FixedSizeList` component, which you rename to `List` for clarity during the import process. You pass it the height of the container, the number of elements, the height of a single element, and the width of the container.

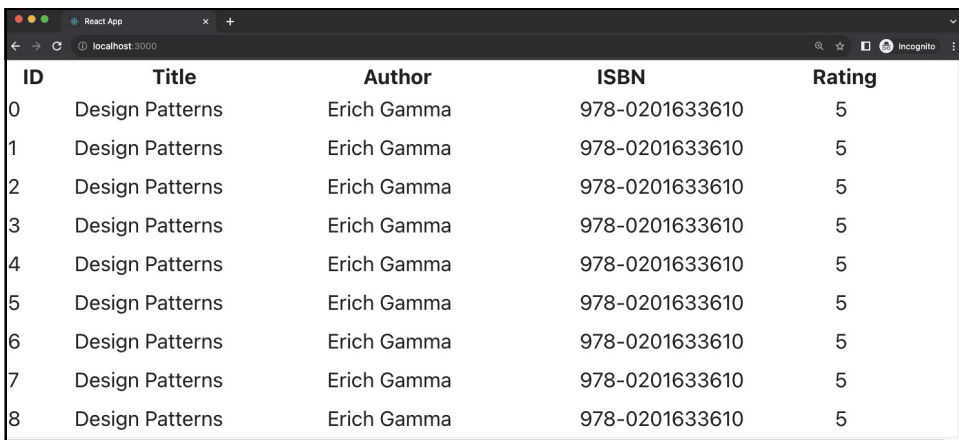
Optionally, you can pass an element or, as in our case, a component to the component via the `innerElementType` prop that includes the elements to be displayed. The `Inner` component renders a table structure and receives the elements to be displayed for virtual scrolling as `children`. The important thing here is that you integrate the `style` property in the JSX structure. In the example, this happens in the `tbody` element, and

`react-window` uses these styles to format the elements in such a way that virtual scrolling works.

Of similar importance is the `Row` component, which you integrate as a child component of the `List` component. It receives the index of the data record to be displayed and also a style object. You can use the index to access the data of the record and the style object to support the correct scrolling behavior of the individual elements.

To make sure the example works correctly, you need to apply some more style definitions, such as the absolute positioning of the `tbody` element or the width of the columns.

When you load the example in the browser, you'll get a view like the one shown in [Figure 19.3](#).



ID	Title	Author	ISBN	Rating
0	Design Patterns	Erich Gamma	978-0201633610	5
1	Design Patterns	Erich Gamma	978-0201633610	5
2	Design Patterns	Erich Gamma	978-0201633610	5
3	Design Patterns	Erich Gamma	978-0201633610	5
4	Design Patterns	Erich Gamma	978-0201633610	5
5	Design Patterns	Erich Gamma	978-0201633610	5
6	Design Patterns	Erich Gamma	978-0201633610	5
7	Design Patterns	Erich Gamma	978-0201633610	5
8	Design Patterns	Erich Gamma	978-0201633610	5

Figure 19.3 Virtual List with “`react-window`”

The interesting aspect of this example is not the display of the table, but the fact that the table has a total of over a million entries and the browser can still display it. The reason this works in this case is that not more than a million DOM nodes are created here, but significantly fewer: the library creates elements only for the visible rows. As soon as you start scrolling or move the scrollbar, `react-window` renders the elements again and swaps the content.

19.7 Summary

In this chapter, you learned more about performance aspects of React. Basically, the library is already very fast, but there are situations in which you need to optimize your application:

- By memoizing functions and objects, you ensure that the structures do not have to be created again for each render operation. For this, you use the `useCallback` and `useMemo` functions.
- Another way you can influence React's rendering behavior is by using the `PureComponent` class, from which you can derive your class component. In this case, React will only re-render your component if the props differ. For props that haven't changed, React reuses the previous instance.
- The counterpart to the `PureComponent` class is the `React.memo` function for function components. It serves the same purpose and ensures that unnecessary component renders are avoided.
- Using `React.lazy`, you can reduce the bundle size of your application by reloading individual components or entire component trees asynchronously.
- By utilizing the fallback content of the `Suspense` components, you can inform your users that components are taking some time to load and display a loading indicator, for example.
- You can use `React.lazy` anywhere in your application. However, this feature makes the most sense for conditional rendering or in conjunction with additional libraries such as `React Router`.
- *Suspense for Data Fetching* integrates the loading of data from server interfaces deeper into your application. You can intercept the load state directly via `Suspense` and show fallback content to your users.
- *Suspense for Data Fetching* requires additional libraries such as `React Query`, `swr`, or `Relay` because the browser's native `Fetch` API is not supported.
- *Virtual scrolling* allows you to get around the problem of very large component trees. Libraries like `react-window` render only the DOM nodes that are really needed for the display. This allows you to display lists with a million or more entries.

In the next chapter, you'll learn how to turn your React app into a progressive web app and provide your users with additional features that go well beyond what ordinary web apps can do.

Chapter 20

Progressive Web Apps

In the development of browser technology, there are always landmarks worth mentioning. Probably one of the best-known paradigms was Ajax, a combination of browser features that existed at the time Ajax was invented that allowed asynchronous communication with a web server without reloading the page. This paradigm laid the foundation for current single-page applications. A similar development, albeit not quite as groundbreaking, can currently be observed with *progressive web apps* (PWAs).

A PWA is basically an ordinary client-side web application that utilizes the modern features of a browser in such a way that the boundaries between a native app in the system and a web application become blurred. The term *progressive* represents the fact that the range of functions expands with the presence of certain browser features, so the basic application can be run in almost any environment. The most important features of a PWA are its ability to work offline and that you can install and use the application on a system like an ordinary application. In this chapter, you'll build the library application into a PWA so that you can achieve these aspects.

20.1 Features of a Progressive Web App

One of the best resources on the topic of PWA is the documentation on Google Developers, which you can find at <https://web.dev/progressive-web-apps/>. As one of the most important browser manufacturers, Google promotes the spread of PWAs, but Microsoft and now Apple also support PWAs in their browsers and systems. On the internet, you can find numerous blog articles, checklists, and instructions on the subject of PWAs. The main features most of these resources discuss in common are as follows:

- **Independence**

A PWA should be able to run on any system that has a browser, regardless of the environment.

- **Adaptable**

The application adapts to the conditions of the respective environment. A classic example of this is the size of the display, but the input methods should also be considered at this point.

- **Reliable**

The PWA must work even if the network connection is poor or nonexistent.

- **Integrated**

The application integrates with the particular environment in which it runs.

- **Secure**

A PWA is always delivered over an encrypted connection.

- **Findable**

A PWA is recognizable as such at a glance. This aspect applies more to the technical level and less to the user level. For example, a search engine must be able to determine whether an application is a PWA.

- **Notification**

The application should be able to notify a user about events in the application.

If your application is to be considered as a PWA, it should meet all of these criteria and features if possible. To determine the degree of fulfillment, numerous checklists and tools are available, some of which we'll share throughout this chapter.

Some criteria for a PWA are met automatically by a React app; for others, you need to customize your app. In this chapter, we'll focus on the fact that the sample application becomes installable and is capable of offline operation and that it is able to send messages to the user.

20.2 Initializing the Application

Not only does Create React App help you to set up an ordinary React app, but it also supports the creation of a progressive web app. To be able to create a PWA, you need to use a special template, which is called `cra-template-pwa` for a JavaScript application and `cra-template-pwa-typescript` for a TypeScript application. Thus you set up your application using the `npx create-react-app library -template cra-template-pwa-typescript` command.

Except for some special features which we'll discuss ahead, the structure is similar to that of an ordinary React application.

20.3 Installability

For a user to install your application on their device, you must meet some requirements:

- The application *has not been installed* yet.
- The application is delivered via *HTTPS*.
- A *web app manifest* exists.
- A *service worker* with a *fetch handler* has been installed.

The first criterion—that the application is not yet installed—goes without saying. If you want to retest the installation process of an already installed application, you can uninstall and reinstall the application. But you should note that—depending on the application—any data generated during its use may be lost.

20.3.1 Secure Delivery of an Application

Numerous browser features that are used in the context of a PWA require the application to be delivered via HTTPS, as otherwise the browser will disable these features. Prominent features from this category include *service workers*, which allow you to achieve offline capability in an application, or the Push API, which enables you to send messages to a browser. For a complete list of browser interfaces that are available only in a secure context, see <http://s-prs.co/v570508>.

When you test a React PWA, there are a few aspects you need to consider. As mentioned, the service worker required for the PWA only works over an encrypted connection and is disabled by default for development operations. The reason is that with the service worker it's possible to cache the source code of the application. For production operation, this means a significant performance advantage. However, during development, caching can cause unwanted side effects. For this reason, the first step is to set up an environment in which you can test your PWA. And here too there is something else to consider: if, as is the normal case, you do not have a certificate signed by a certificate authority, your browser will not accept such a self-signed certificate as secure. As a consequence, some browser features will still not be available.

To facilitate web application testing, the secure context applies to all addresses except localhost. This means that you don't have to use HTTPS for local testing and the features still work. In the following sections, you'll learn how to generate a self-signed certificate to test your application on a remote web server.

Creating and Installing a Certificate

Generating a certificate file is usually done via a tool like OpenSSL on the command line. It accepts multiple and generates a key file and a certificate file that you can use to encrypt communication between the client and the server. [Listing 20.1](#) shows an example of calling the OpenSSL tool in the command line of a macOS system:

```
openssl req \
  -newkey rsa:2048 \
  -x509 \
  -nodes \
  -keyout key.pem \
  -out cert.pem \
  -subj /CN=localhost \
  -reqexts SAN \
```

```
-extensions SAN \
-config <(cat /System/Library/OpenSSL/openssl.cnf \
    <(printf '[SAN]\nsubjectAltName=DNS:localhost')) \
-days 365
```

Listing 20.1 Generating a Certificate and Key

Especially if you use Chrome, you need to make sure that you set a subject alternative name (SAN) in the certificate; otherwise the browser will mark the certificate as invalid. If you run the script from [Listing 20.1](#) in the root directory of your application, the result consists of two files: *key.pem* and *cert.pem*. Currently, the certificate is an untrusted self-signed certificate.

If you delivered your application with this certificate, Chrome would tell you that the page is not secure. By clicking on the warning sign, you'll get more information that will show you that the certificate is invalid. Another click on the certificate will show you the details (see [Figure 20.1](#)), from which you can get more information about this certificate—for example, that the certificate has not been verified by a third party.

For a local test, you can work around this by installing the certificate and trusting it. The certificate can be installed by dragging and dropping it onto your file system and then double-clicking on it to install it. In the certificate management of your system, you then have the option to trust the certificate. [Figure 20.2](#) shows what this looks like in a macOS system.

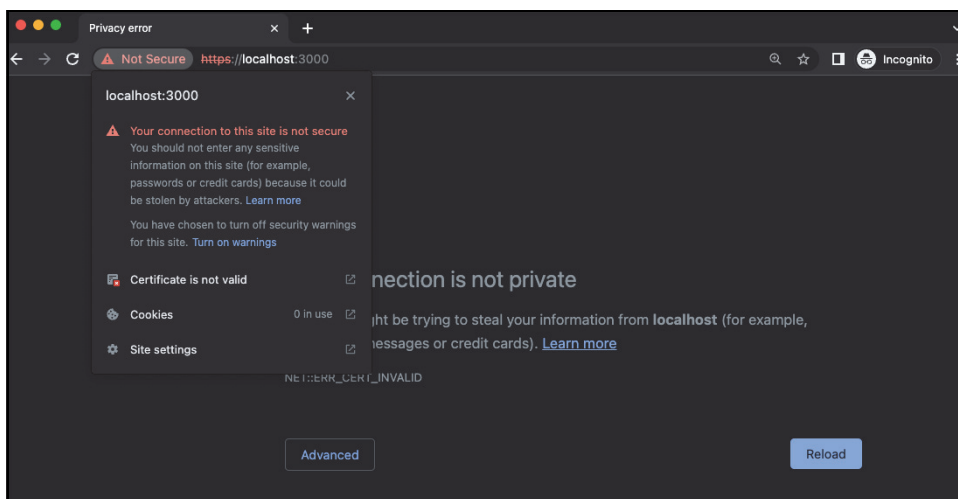


Figure 20.1 Security Warning for Invalid Certificate

Once you've implemented this setting, you'll need to restart your browser to apply the updated information. Then you can communicate with your local web server over HTTPS without any interruption.

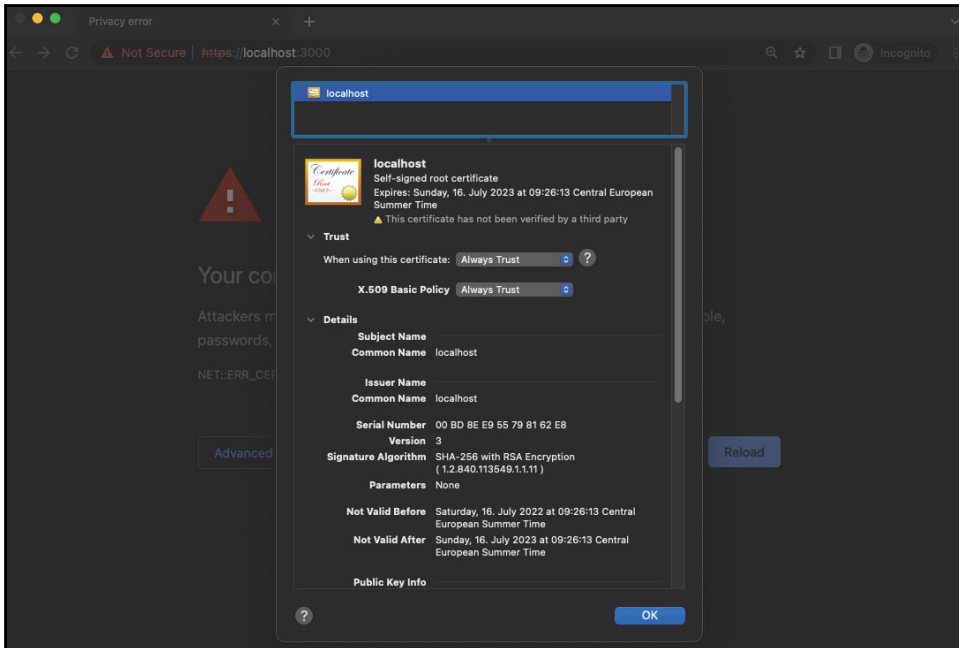


Figure 20.2 Trusting the Self-Signed Certificate

Delivery of the Application Over a Secure Connection

You can already deliver your application with the current state over HTTPS. The first step is to use the `npm run build` command to build your application for production use. This command creates the built version in the `build` directory in your application. You normally deliver this directory via a web server. However, for testing the PWA, you can also use the `http-server` package and deliver the application directly.

You use the `npx http-server ./build -S` command to start the web server. The `-S` option makes sure that the HTTPS protocol is used instead of the default HTTP. The `http-server` searches in the current directory for the `key.pem` and `cert.pem` files to establish the encrypted connection. Instead of the `not secure` display, you'll now see a small padlock to the left of the address bar. This means that the application is delivered through an encrypted connection.

20.3.2 The Web App Manifest

The second element that's a prerequisite for the installability of a PWA is the existence of a *web app manifest*. This is a file that defines the behavior of the installable PWA in more detail. The web app manifest shouldn't be confused with the older cache manifest. The cache manifest was an attempt to make web applications capable of offline operation. If you create your React app using Create React App, a `manifest.json` file will automatically be created in the `public` directory, but it meets the requirements for the

installability of the application only partially. In [Table 20.1](#), you can see the properties of the web app manifest.

Property	Meaning
background_color	Background color of the application. Used to set the background before the styles get loaded.
description	Description of the application.
dir	Text direction used in the application. Possible values are the default auto, rtl or ltr.
display	This property defines the way the application is displayed.
icons	The icons property contains the descriptions of the image files that will be used for the application after installation, for example.
lang	Language or locale of the application.
name	The name of the application.
orientation	Device orientation of the application.
prefer_related_applications	If this property is set to true, a native app is recommended instead of the web app.
related_applications	An array of related native applications that can be used instead of the web application.
scope	This property sets the web pages that can be displayed within the manifest. When navigating outside this area, the page will be displayed in the normal browser window.
short_name	Shortened name of the application.
start_url	The entry point into the application.
theme_color	Main color of the application; partially used by the operating system for display.

Table 20.1 Features of the Web App Manifest

To install your application on your system, you need to make some adjustments to the existing manifest file. However, you should always do this whether you’re interested in creating an installable PWA or not in order to adapt the description to the circumstances of your application.

```

{
  "short_name": "Library",
  "name": "Books Management",
  "description": "An application that lets you manage your books",
  "icons": [
    {
      "src": "favicon.ico",
      "sizes": "64x64 32x32 24x24 16x16",
      "type": "image/x-icon"
    },
    {
      "src": "logo192.png",
      "type": "image/png",
      "sizes": "192x192"
    },
    {
      "src": "logo512.png",
      "type": "image/png",
      "sizes": "512x512"
    }
  ],
  "start_url": ".",
  "display": "standalone",
  "theme_color": "#000000",
  "background_color": "#ffffff"
}

```

Listing 20.2 Customized Manifest File of the Application (public/manifest.json)

As you can see in [Listing 20.2](#), the `short_name` and `name` properties have been adjusted for the application and a description has been added via the `description` property. Another requirement for the installability of the application is the presence of an icon, which has a height and width of at least 144 pixels. You can use the `icons` property to specify an array of icon objects. These have the `src`, `sizes`, and `type` properties. The `src` property defines the path to the file within the *public* directory. The `sizes` property is a string containing the resolutions for which the file is to be used. The individual size specifications are separated from each other by spaces. Finally, you use `type` to specify the type of the file. To meet the specifications in the manifest file, you place two image files in the *public* directory of your application with a height and width of 192 and 512 pixels, respectively, and name them *icon192.png* and *icon512.png*, respectively. Alternatively, you can use the default files Create React App has provided for you.

20.3.3 Service Worker in the React Application

The last step before installing your application consists of activating the service worker. Again, Create React App has already made preparations for you. The *serviceWorker.ts* file contains a preconfigured service worker, but it's disabled by default. If you take a look at the *index.tsx* file that Create React App generates for you, you'll see a comment block at the end of the file informing you that you should use the `register` method instead of the `unregister` method if you want to have a faster app that's capable of offline operation. In [Listing 20.3](#), you can see the customized source code of the *index.tsx* file that makes sure that the service worker is activated:

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import App from './App';
import * as serviceWorkerRegistration from ←
  './serviceWorkerRegistration';
import reportWebVitals from './reportWebVitals';

const root = ReactDOM.createRoot(
  document.getElementById('root') as HTMLElement
);
root.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
);

// If you want your app to work offline and load faster, you can change
// unregister() to register() below. Note this comes with some pitfalls.
// Learn more about service workers: https://cra.link/PWA
serviceWorkerRegistration.register();

// If you want to start measuring performance in your app,
// pass a function
// to log results (for example: reportWebVitals(console.log))
// or send to an analytics endpoint. Learn more: https://bit.ly/CRA-vitals
reportWebVitals();
```

Listing 20.3 Initial File of the Application with Activated Service Worker (*src/index.tsx*)

The *serviceWorker.ts* file makes sure that the service worker is active only in the production environment. So at this point you need to build your application again using the `npm run build` command and deliver it using `npm run http-server ./build`. If you then switch to the browser and go to `http://localhost:8080/`, you'll have the option to install the application.

20.3.4 Installing the Application

In Chrome, you can install an app that meets the PWA requirements via a separate menu item. [Figure 20.3](#) shows the **Install book management....** item in the lower part of the menu. Clicking on this item starts the installation process.

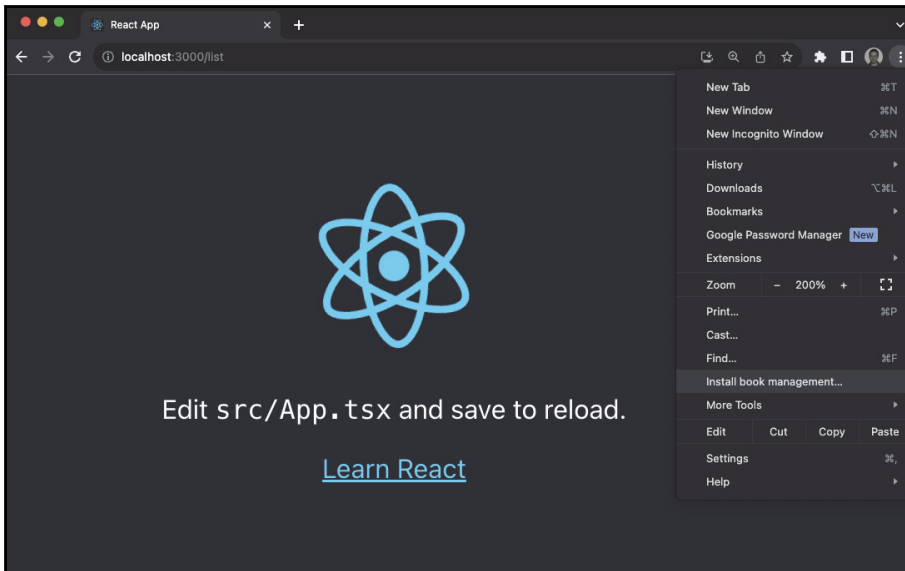


Figure 20.3 Installation of the Application

The effect of installing a PWA is that it gets its own place in the Start bar of your system and in the app switcher as well as an icon to launch it. Before the application is installed, you must confirm this in a separate dialog box that looks similar to the one in [Figure 20.4](#).

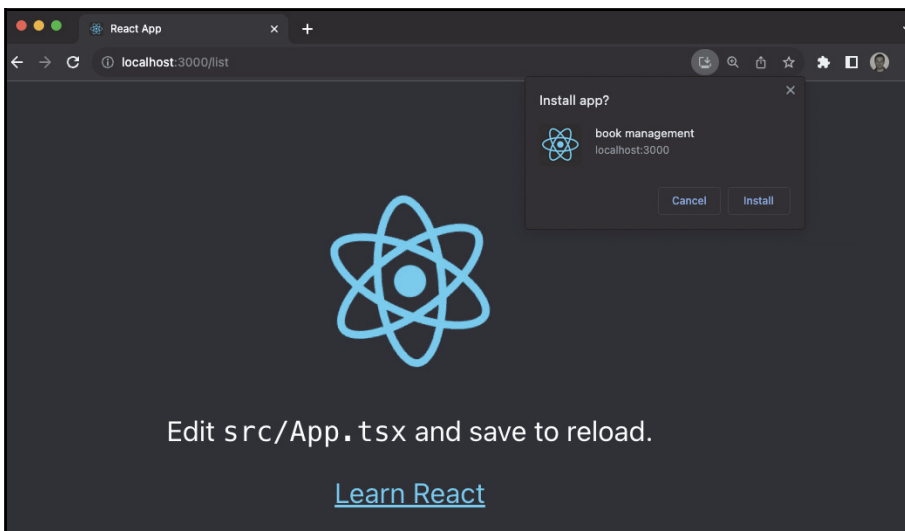


Figure 20.4 Confirmation of the Installation

Configuring the `display` property in the manifest file means that the PWA looks different after launching as an installed application than when it runs in the regular browser. The `standalone` value ensures that the menu and address bars of the browser are hidden and that the application looks like a regular application of the system. [Figure 20.5](#) shows the current state of the application.

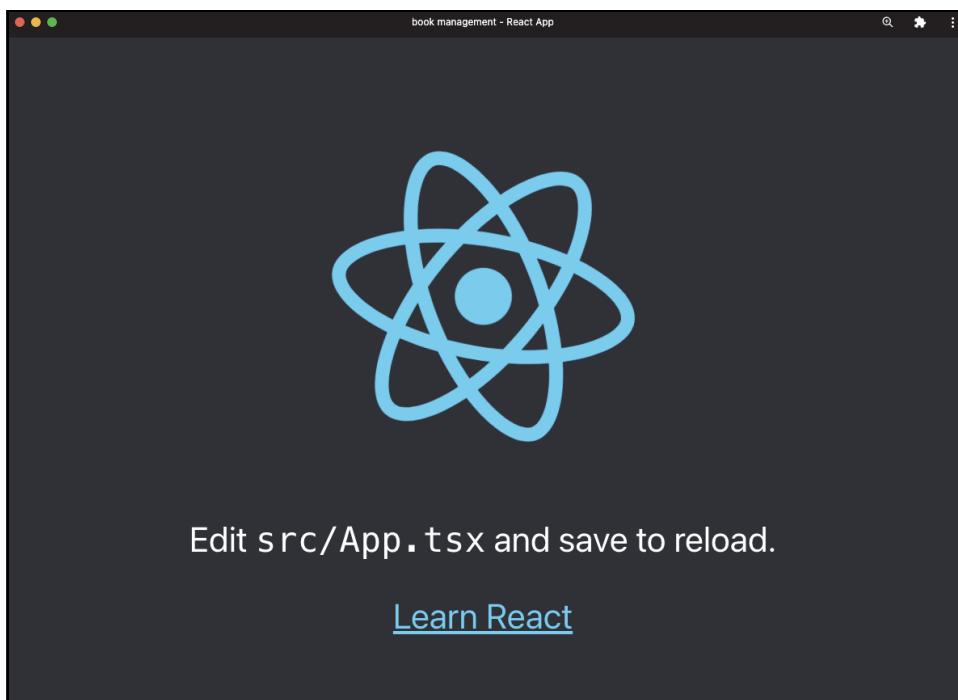


Figure 20.5 View of the Installed PWA

Because the application still runs in the browser and only the controls are hidden, you still have the option to open your browser's developer tools via the context menu or keyboard shortcut.

20.3.5 Asking the Users

A browser feature enables you to actively ask users to install the application. Currently, the `beforeinstallprompt` event, which is the basis for this feature, is still in *working draft* status and is only supported by Chrome. This means that it isn't available in many browsers. However, with this extension of your application you follow the basic concept of PWAs: new browser features extend the functionality of an application; if these features aren't available on the current platform, then the application will still function without the respective extension.

In the simplest case, you register an event listener to the `beforeinstallprompt` event in the `index.jsx` file of your application. The basic implementation of this event is that the

installation dialog is displayed directly to the users if the prerequisites for the installation are met. As with all active requests to users, you should also proceed with caution here. After all, if a person visits the application for the first time, it's probably a bit too early to ask for the application to be installed before they even log in. You should ensure that your application really adds value, making it attractive for your users to install it.

The core of the implementation is a custom hook, which you use to register for the event. The custom hook makes sure that the default action—that is, the request—is not executed and the event is returned. You then execute the `prompt` method of the event object responsible for the request inside the `List` component. In the first step, you create a file named `useAddToHomeScreen.ts` in the `src` directory that contains the source code of the hook:

```
import { useEffect, useState } from 'react';

export interface BeforeInstallPromptEvent extends Event {
  platforms: string[];
  userChoice: Promise<{
    outcome: 'accepted' | 'dismissed';
    platform: string;
  }>;
  prompt(): Promise<void>;
}

export default function useAddToHomescreen(): BeforeInstallPromptEvent | null {
  const [prompt, setPrompt] = useState<
    BeforeInstallPromptEvent | null
  >(null);
  useEffect(() => {
    const handleBeforeInstallPrompt = (e: Event) => {
      e.preventDefault();
      setPrompt(e as BeforeInstallPromptEvent);
    };
    window.addEventListener(
      'beforeinstallprompt', handleBeforeInstallPrompt
    );
    return () =>
      window.removeEventListener(
        'beforeinstallprompt',
        handleBeforeInstallPrompt
      );
  }, []);
```

```
if (prompt) {  
  return prompt;  
}  
return null;  
}
```

Listing 20.4 “AddToHomeScreen” Hook (src/useAddToHomeScreen.ts)

In [Listing 20.4](#), you first create an interface for the event structure since it’s not yet part of the current TypeScript definition for the browser events. The subsequent implementation of the hook is a custom hook consisting of an effect hook and a state hook. Inside the effect hook, you define the actual event handler for the `beforeinstallprompt` event.

What’s important at this point is that you use `preventDefault` to ensure that users do not immediately receive the request to install. Instead, you store the event object in the state of the custom hook. By returning a function in the effect hook, you make sure that it’s used as a teardown routine in which you can remove the event handler again.

The return value of the custom hook is either the event object, if the event was triggered, or the value `null`. This way, you can make sure that the code can work in any browser.

In the next step, you include the custom hook in the `App` component and pass the return value to the `List` component. The modifications to the `App` component required for this are shown in [Listing 20.5](#):

```
import React from 'react';  
import './App.css';  
import useAddToHomescreen from './useAddToHomeScreen';  
import List from './List';  
  
const App: React.FC = () => {  
  const prompt = useAddToHomescreen();  
  return <List prompt={prompt} />;  
};  
  
export default App;
```

Listing 20.5 Integration of the “AddToHomeScreen” Hook into the “App” Component (src/App.tsx)

This allows you to make the installation request in the `List` component in another effect hook. [Listing 20.6](#) shows the required modifications to the `List` component. In the query, you check that the return value of the `useAddToHomeScreen` function doesn’t have the value `null`:

```

import React, { useEffect } from 'react';

import { BeforeInstallPromptEvent } from './useAddToHomeScreen';

type Props = {
  prompt: BeforeInstallPromptEvent | null;
};

const books = [
  {
    id: '1',
    title: 'JavaScript - The Comprehensive Guide',
    isbn: '978-3836286299',
  },
  {
    id: '2',
    title: 'Clean Code',
    isbn: '978-0132350884',
  },
  {
    id: '3',
    title: 'Design Patterns',
    isbn: '978-0201633610',
  },
];

const List: React.FC<Props> = ({ prompt }) => {
  useEffect(() => {
    if (prompt) {
      prompt.prompt();
    }
  }, []);

  return (
    <table>
      <thead>
        <tr>
          <th>Title</th>
          <th>ISBN</th>
        </tr>
      </thead>
      <tbody>
        {books.map((book) => (
          <tr key={book.id}>

```

```
        <td>{book.title}</td>
        <td>{book.isbn}</td>
      </tr>
    )})
  </tbody>
</table>
);
};
```

```
export default List;
```

Listing 20.6 Asking the User if the Application Should Be Installed (src/List.tsx)

To test the new feature of your application, you need to rebuild your application and deliver the resulting *build* directory encrypted with a web server. When you then open your app in Chrome, you will be asked if you want to install the app.

The prerequisite for this is that you have either not installed the application previously or that you have already uninstalled it. If the request doesn't display as expected, this may be because an old state of the application is in the service worker cache. You can empty the cache in Chrome's developer tools via the **Application** tab and the **Clear storage** option.

Once it's installed, the users of your application can use it like a native application on their system. The target system can be both a desktop computer and a smartphone. Another feature of PWAs that goes hand in hand with installability is offline capability, which we'll address in the following section.

20.4 Offline Capability

Installing an application suggests to the user that the entire source code has been downloaded and the application can be used on the system, whether or not there's an internet connection. But if you switch to the installed application and turn off the network connection in the developer tools, you'll quickly notice that your application's offline capability isn't yet available.

This problem can be solved by integrating *Workbox*, a tool for configuring service workers. This project is now used by Create React App, but it can't be configured to its full extent in the default configuration. The developers are currently working on an appropriate solution. Until then, let's look at a solution that allows you to integrate your own service worker implementation based on Workbox. This allows you not only to make your application offline-capable, but also to get better control over the behavior of the service worker.

20.4.1 Integrating Workbox

First, you need to install a set of dependencies in your application. To configure your service worker, you use the `npm install --save-dev workbox-build workbox-cli workbox-sw` command to install the following packages:

- **workbox-build**

This package provides support for creating the service worker configuration.

- **workbox-cli**

This is the command line tool of Workbox.

- **workbox-sw**

This is the Workbox library itself.

To make the development process for your PWA a bit more convenient, you also install the `http-server` package you used to deliver the application in test mode. To do this, you use the `npm install --save-dev http-server` command. After installing the packages, you create some NPM scripts that provide the most important operations. In [Listing 20.7](#), you can see the extension of the `scripts` section of your `package.json` file:

```
{
  ...
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build && npm run build-sw",
    "test": "react-scripts test",
    "eject": "react-scripts eject",
    "build-sw": "node ./src/sw-build.js",
    "serve": "http-server ./build",
    "run-pwa": "yarn build && yarn serve",
    "copy-wb-lib": "workbox copyLibraries public/"
  },
  ...
}
```

Listing 20.7 Extension of the `package.json` File with Additional NPM Scripts

The following list contains a short description for each of the newly added scripts with hints on how they can help you work with your PWA:

- **build**

The `build` script has been extended to run the `build-sw` script.

- **build-sw**

This script creates the service worker using Workbox. The `sw-build.js` file contains the required commands.

- **serve**

This script provides the `build` directory using the `http-server` package.

- **run-pwa**

The `run-pwa` script first builds the PWA and then delivers it.

- **copy-wb-lib**

To initialize the application, you use the `copy-wb-lib` script.

First, you run the `copy-wb-lib` script via the `npm run copy-wb-lib` command. Using this script, the required Workbox files are copied to a subdirectory of the *public* directory in your application. This allows you to avoid having to download the corresponding files from the internet via a CDN.

The copied library files are from the Workbox package. For this reason, you don't need to integrate it into your application's version control system and can ignore the Workbox directory in the *public* directory. If you use Git for version control, you can ignore the Workbox directory by extending the `.gitignore` file of your application as shown in [Listing 20.8](#):

```
...
public/workbox*
```

Listing 20.8 Ignoring the Copied Workbox Files in the public Directory (`.gitignore`)

Because you create your own service worker via Workbox, you'll need to make some adjustments to the loading process in the files prepared by Create React App. In particular, this affects the `serviceWorkerRegistration.ts` file in the *src* directory of your application. There you change the `swUrl` variable from `service-worker.js` to `sw.js`, as in [Listing 20.9](#):

```
...
const swUrl = `${process.env.PUBLIC_URL}/sw.js`;
...
```

Listing 20.9 Adjusting the Service Worker URL (`src/serviceWorkerRegistration.ts`)

In the next step, you need to create the service worker. Workbox provides two options to do that: you can use either the `generateSW` or the `injectManifest` method. Using `generateSW` is the standard way, and this configuration is sufficient for some use cases.

The `injectManifest` method that we'll use in the example provides a better control over the service worker, but it's a bit more complex to configure. To build the service worker using Workbox, you need to create the `sw-build.js` file in the *src* directory. In [Listing 20.10](#), you can see the content of this file:

```
const workboxBuild = require('workbox-build');
workboxBuild.injectManifest({
  swSrc: 'src/sw.js',
  swDest: 'build/sw.js',
```

```

    globDirectory: 'build',
    globPatterns: ['**/*.{js,css,html,png,woff,woff2,json,ico}'],
  });

```

Listing 20.10 Configuration for the Offline Capability of the Application (src/sw-build.js)

You pass a configuration object to the `injectManifest` method, which prepares the service worker accordingly. The `swSrc` and `swDest` properties specify the source and destination files of the service worker, respectively. By using the `globDirectory` property, you specify the base directory to search for files to cache. Finally, the `globPatterns` property specifies which files are to be stored in the cache. The actual configuration of the service worker is done in the `sw.js` file in the `src` directory, as you can see in [Listing 20.11](#):

```

importScripts('/workbox-v6.5.3/workbox-sw.js');

if (workbox) {
  workbox.setConfig({
    modulePathPrefix: '/workbox-v6.5.3/',
  });

  workbox.precaching.precacheAndRoute(self.__WB_MANIFEST);

  workbox.routing.registerRoute(
    /\.?(?:png)$/,
    new workbox.strategies.CacheFirst({
      cacheName: 'images',
      plugins: [
        new workbox.expiration.ExpirationPlugin({
          maxEntries: 60,
          maxAgeSeconds: 30 * 24 * 60 * 60,
        }),
      ],
    })
  );
} else {
  console.log('Workbox could not be loaded. No Offline support');
}

```

Listing 20.11 Configuration of the Service Worker (src/sw.js)

In [Listing 20.11](#), you can see the source code of the `sw.js` file that's responsible for configuring the service worker. For this purpose, the `workbox` object is first loaded from the `workbox-sw.js` file. You can either obtain this file from a CDN or use the Workbox CLI as in the example, copy the files to the `public` directory, and then refer to the files stored

there. The `copyLibraries` command of the Workbox CLI creates a new subdirectory with the number of the Workbox version that's being used.

Using the `importScript` function, you load the `workbox-sw.js` file that provides the `workbox` object. Then you set the `modulePathPrefix` property to the path where the files are located. The `precacheAndRoute` method prepares the precaching process. The call of the `registerRoute` method shows a concrete example of service worker caching as it's carried out by Workbox.

For all files ending in `.png`, the cache-first strategy is used. This means that when a request is made for such a file, the service worker cache is searched first and only then is the server requested. The validity of the cache entries is set to 30 days, while the number of entries is limited to 60.

The `build-sw.js` and `sw.js` files are JavaScript files, which cause error messages during the build process of your application. For this reason, you exclude these two files from the build by adding them to the `exclude` property, as you can see in [Listing 20.12](#):

```
{
  ...
  "include": ["src"],
  "exclude": ["src/sw.js", "src/sw-build.js"]
}
```

Listing 20.12 Excluding the Workbox Files (tsconfig.json)

If you now execute the `npm run run-pwa` command on the command line, the application will be built and delivered encrypted via the web server, and you can test the newly acquired offline capability. When you open your browser's developer tools and load the application, you can see from the console output that the application has been loaded into the cache, as shown in [Figure 20.6](#).

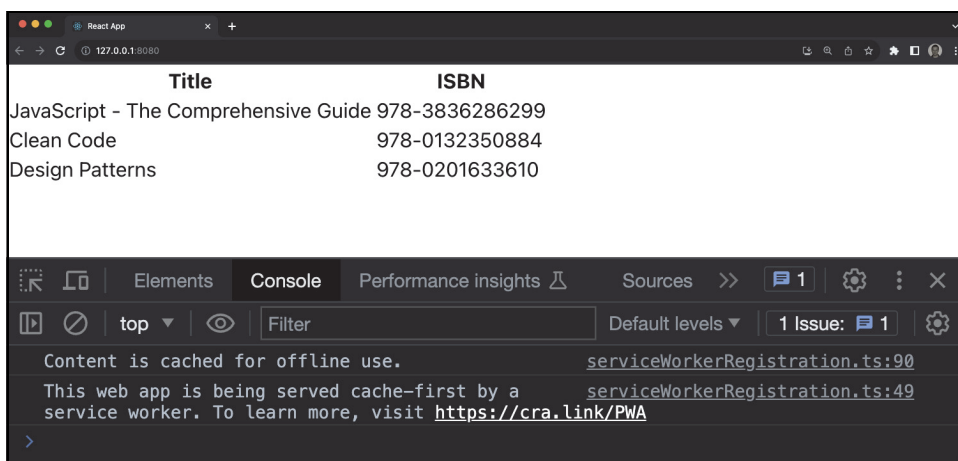


Figure 20.6 Application Written to the Service Worker Cache

To be able to test whether your application is really capable of working offline, you need to switch off the network via the **Network** tab. Alternatively, you can go to the **Application** tab, select the **Service Worker** menu item, and check the **offline** option. If you now reload your application, it will be completely loaded from the cache. Again, the console output in the browser provides information about what's happening, as you can see in [Figure 20.7](#).

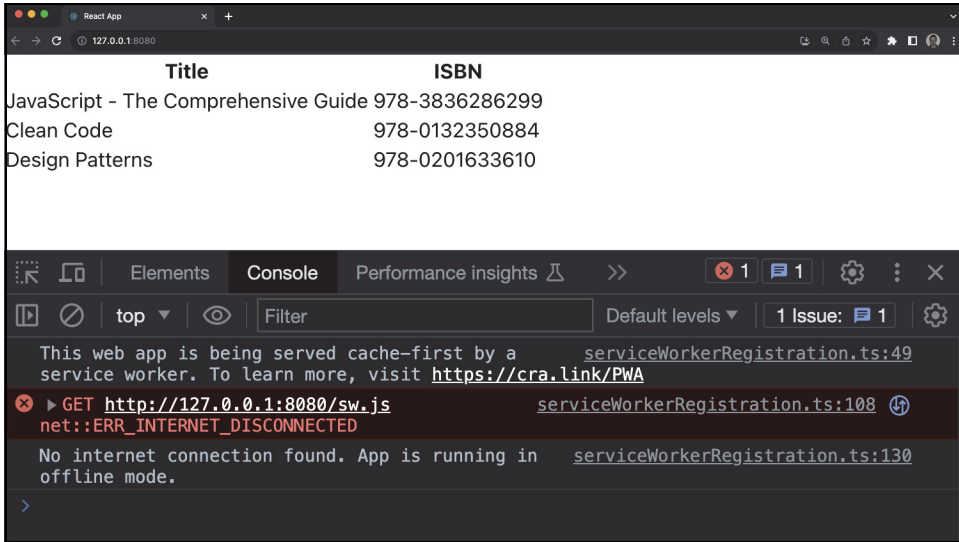


Figure 20.7 Console Output for an Application from the Cache

However, there's more to an application's offline capability than just storing static content in the browser cache. In the next section, you'll see a solution for the dynamic data of the application.

20.4.2 Handling Dynamic Data

If there's no connection between the browser and the server of your application, the browser can load the application from the cache, but the display of dynamic data doesn't work. In this section, you'll create a solution to this problem so that users can fully use your application offline.

Dealing with dynamic data is a complex field, especially when it comes to write accesses. If users are offline and generate data, you need to synchronize this data as soon as they're connected again. But then a problem can occur: in the meantime, other users also have write access to the server. In this case, a conflict arises, which must be resolved. As with caching, you can follow different strategies. These range from "the last write wins" to complex algorithms for the automated merging of information. You must decide which strategy you want to pursue on a case-by-case basis for each application.

To enable your users to use your application offline, you need to make sure that the data is stored locally.

Preparing the Application

First, you prepare a simple server interface and rebuild the `List` component to get its data from the server. Then you make sure that the data is obtained from the local data store in the offline state.

For the server, you install the `json-server` package using the `npm install json-server` command. The `data.json` file in the root directory of your application serves as a data source, the contents of which are shown in [Listing 20.13](#):

```
{
  "books": [
    {
      "id": 1,
      "title": "JavaScript - The Comprehensive Guide",
      "author": "Philip Ackermann",
      "isbn": "978-3836286299",
      "rating": 5
    },
    {
      "id": 2,
      "title": "Clean Code",
      "author": "Robert Martin",
      "isbn": "978-0132350884",
      "rating": 4
    },
    {
      "id": 3,
      "title": "Design Patterns",
      "author": "Erich Gamma",
      "isbn": "978-0201633610",
      "rating": 5
    }
  ]
}
```

Listing 20.13 Data Source of the Server (`data.json`)

For convenient server startup, you add a backend script to the `scripts` section of your `package.json` file as shown in [Listing 20.14](#):

```
{
  ...
  "scripts": {
    ...
    "backend": "json-server -p 3001 -w data.json"
  }
}
```

```
...
}
```

Listing 20.14 “backend” Script in the package.json File (package.json)

Having implemented these adjustments, you can start your backend via the `npm run backend` command and run this process in the background in a separate command line.

In the next step, you change the `List` component in such a way that it gets your display data from the server:

```
import React, { useEffect, useState } from 'react';
import { Book } from './Book';

const List: React.FC = () => {
  const [books, setBooks] = useState<Book[]>([]);
  useEffect(() => {
    fetch('http://localhost:3001/books')
      .then((response) => response.json())
      .then((data) => setBooks(data));
  }, []);

  return (
    <table>
      <thead>
        <tr>
          <th>Title</th>
          <th>ISBN</th>
        </tr>
      </thead>
      <tbody>
        {books.map((book) => (
          <tr key={book.id}>
            <td>{book.title}</td>
            <td>{book.isbn}</td>
          </tr>
        ))}
      </tbody>
    </table>
  );
};
```

```
export default List;
```

Listing 20.15 Implementation of the “List” Component with Server Connection (src/List.tsx)

On this basis, you can now ensure that dynamic queries also become offline-capable.

Storing the Data Locally

There are numerous solutions available for storing data locally. These range from the capabilities of the browser itself to client-server databases that synchronize automatically. A widely used approach is to use the browser's own *IndexedDB*, an indexed database that allows for the quick retrieval of data records.

Because the interface of IndexedDB is not among the most convenient ones, it's recommended to use an abstraction layer such as Dexie at this point. *Dexie* is a framework-independent open-source library implemented in TypeScript. Using `npm install dexie`, you can install the library. In this example, you'll store the data for the books locally in the browser. This means that for this purpose, you need to create an object that encapsulates access to the IndexedDB. You save the Dexie object in the *src* directory and create the *bookDatabase.ts* file there. The contents of this file are shown in [Listing 20.16](#):

```
import Dexie from 'dexie';
import { Book } from './Book';

class BookDatabase extends Dexie {
  public books: Dexie.Table<Book, number>;

  public constructor() {
    super('BookDatabase');
    this.version(1).stores({
      books: 'id,title',
    });
    this.books = this.table('books');
  }
}

export default new BookDatabase();
```

Listing 20.16 Access to IndexedDB (*src/bookDatabase.ts*)

The responsibility for where the data is read from lies at the point of communication with the server. In the example, this is the *List* component. You must first query whether the client is in online mode, then obtain the data from the server or send the data to the server. If the client doesn't have an internet connection, then you can access the IndexedDB instance of the browser via Dexie. The only difficulty with write accesses consists of synchronizing the information once a connection is reestablished. You can obtain this information from the *online* event of the browser.

Ahead, you'll implement the *List* component. [Listing 20.17](#) shows the customized source code of the component:

```
import React, { useEffect, useState } from 'react';
import { Book } from './Book';
```

```

import bookDatabase from './bookDatabase';

const List: React.FC = () => {
  const [books, setBooks] = useState<Book[]>([]);

  async function fetchData() {
    let data: Book[];
    if (navigator.onLine) {
      const response = await fetch('http://localhost:3001/books');
      data = await response.json();
      bookDatabase.books.clear();
      bookDatabase.books.bulkAdd(data);
    } else {
      data = await bookDatabase.books.toArray();
    }
    setBooks(data);
  }

  useEffect(() => {
    fetchData();
  }, []);

  return (
    <table>
      <thead>
        <tr>
          <th>Title</th>
          <th>ISBN</th>
        </tr>
      </thead>
      <tbody>
        {books.map((book) => (
          <tr key={book.id}>
            <td>{book.title}</td>
            <td>{book.isbn}</td>
          </tr>
        ))}
      </tbody>
    </table>
  );
};

export default List;

```

Listing 20.17 Communication via IndexedDB (src/List.tsx)

The `fetchData` function encapsulates communication with the server. In the call of the `fetch` method, you include a query for the online state of the browser. If a connection exists, you can communicate with the server as usual and write the data to IndexedDB after a successful response. To avoid conflicts at this point, you clear the `books` table by calling the `clear` method beforehand. If the browser is in offline mode, you read the data from the `books` table and use the `toArray` method to convert the result into an array that you pass to the `setBooks` function.

If you run the `npm run-pwa` command again at this point and reload your application once, the book data will be written to IndexedDB and you can use the application in offline mode. You can verify that everything worked by opening the developer tools of your browser and viewing the contents of IndexedDB. [Figure 20.8](#) shows an example of such a view.

Besides the debugger that you can use to debug your application, Chrome provides another helpful feature that you can use to develop your PWA, which you'll learn about in the next section.

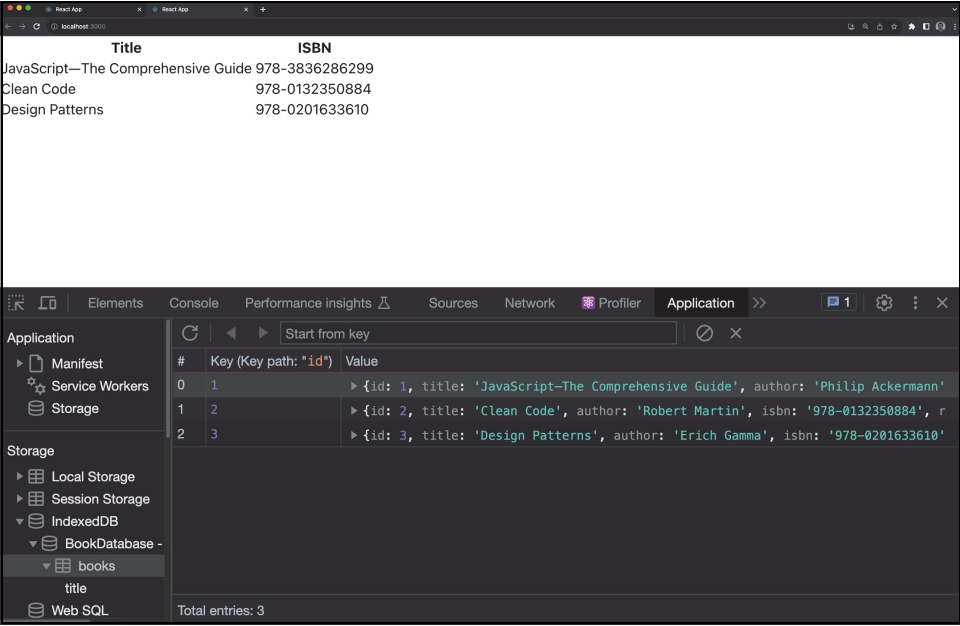


Figure 20.8 Content of IndexedDB

20.5 Tools for Development

Chrome has a built-in tool called *Lighthouse* that is an analysis tool for websites. You can use Lighthouse for different aspects of your application, such as performance or SEO factors, but also for PWA compatibility. Lighthouse then runs a series of checks and calculates a value that indicates how much PWA compatibility is in your application.

To use Lighthouse, you need to open the developer tools of the browser and go to the **Lighthouse** tab, where you can select the different audit areas. For our application, only the **Progressive Web App** section is interesting at the moment. If you then click **Analyze Page Load**, the application scan will start. [Figure 20.9](#) shows a part of the output for the current application.

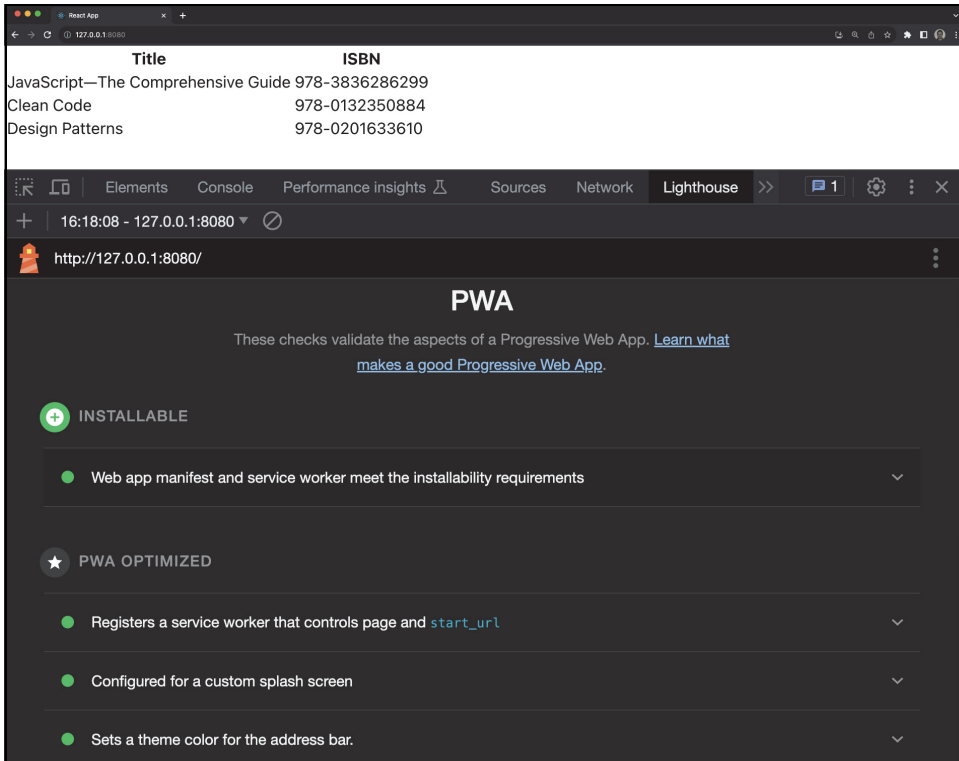


Figure 20.9 Output from Lighthouse

As you can see, you'll get more details and explanations for each message.

In addition to this tool, numerous other resources on the topic of PWA exist on the internet. Most of them are not specific to React but provide guidance on how you can further improve your application. One of the most popular examples is Google's PWA checklist, which can be found at <https://web.dev/pwa-checklist/>.

20.6 Summary

In this chapter, you've learned how to turn your React app into a progressive web app:

- If you build your application using Create React App, basic elements such as the manifest file and a service worker will already be created for you.

- To make your application installable, you must deliver it over an encrypted connection, enable the service worker, and place appropriate icons in the manifest file.
- You can prompt your users to install it using the `beforeinstallprompt` event. This event is triggered when the application meets all installability requirements.
- *Workbox* allows you to configure your service worker in order to store the application's static content in the browser. This enables your users to use the application offline.
- However, there is much more to the offline capability of an application than just storing the static content. You can store dynamic data locally via IndexedDB in the browser.
- The `onLine` property and the `online` event enable you to determine if there is a connection to the internet.
- *Lighthouse* provides a tool to check your PWA. This tool is integrated into the developer tools of Chrome.

In the next chapter, you'll go one step further and bring your React app to mobile devices using React Native.

Chapter 21

Native Apps with React Native

So far, you've designed and implemented your React application to run in a web browser. However, the modular design of React also allows it to run in other environments, such as on the command line or natively on mobile devices. The core of the library remains the same in all environments. What this means to you is that as you develop, you can be confident that the core concepts are available in any environment and you only need to adjust to the changing rendering.

In this section, you'll learn how to implement an application in React Native and install and use it on a mobile device.

21.1 The Structure of React Native

The concept behind React Native is that a React app should be rendered as a native app for Android and iOS. This makes it possible to distribute such an app via Apple's App Store and the Google Play Store. Native apps also have more access rights to native interfaces. The idea of writing native apps with web technologies is not new. The best-known framework in this area is *Cordova*, in which a web application is executed in what's called a *webview*. This means that the combination of HTML, CSS, and JavaScript is packaged together, but still executed in a reduced browser context.

React Native takes a different approach at this point and translates the JSX structures into native elements. This means that you no longer use `div`, `button`, and `input` elements, but other elements that have a native equivalent. Depending on the environment, the native elements are either written in Objective-C and Swift (in the case of an iOS app) or in Java or Kotlin (for Android). The JavaScript source code of the application—that is, the actual business logic—runs in a separate JavaScript process. In most cases, JavaScriptCore, the JavaScript engine from Apple's Safari browser, is used here. To support modern features, React Native uses Babel with a set of preconfigured presets. The advantage of React Native is that you can develop native apps and don't have to deal with the programming language of the target environment.

21.2 Installing React Native

There are several variants of first steps with React Native. For example, you can develop directly with Expo without a heavyweight development environment like Xcode for

iOS or Android Studio for Android. However, for running your app in a simulator, installing these development environments is still useful. Another alternative is to use the React Native CLI; however, this requires a development environment. If you haven't yet installed one on your system, you should allow some time for the installation and configuration.

In this chapter, we'll work with *Expo*, which provides services for the build and deployment processes of an app. To start the development process of your app, you need the Expo CLI, which you can install via `npm install -g expo-cli`. This makes the `expo` command available in the command line. Then, with `expo init --npm`, you initialize your project in the same way as Create React App. The `--npm` option makes sure that NPM is used as the package manager; otherwise, Expo will resort to Yarn as the default.

First, Expo prompts you for the name of your application. For the example application, we use the name *library*. When you initialize your project, Expo offers a number of different templates, which provide the basic structure:

- **Blank**
Minimal configuration of the application, which allows you to start right away
- **Blank (TypeScript)**
Like **Blank**, but with TypeScript preconfigured
- **Tabs (TypeScript)**
Multiple views and tabs among which you can navigate using `react-navigation`
- **Minimal**
Minimal configuration for an application

For the following examples, choose **Blank (TypeScript)**. After this selection, Expo downloads all the required dependencies and creates the initial directory and file structure of your application.

21.2.1 Project Structure

The directory structure created by Expo differs significantly from the standard structure of a React application. In the root directory of the newly created application, you'll find a number of files and directories. [Table 21.1](#) explains their purpose.

File or Directory name	Purpose
<i>.expo-shared</i>	This directory contains the optimized assets of the application.
<i>assets</i>	The <i>assets</i> directory contains assets such as images, videos, and other media files.
<i>node_modules</i>	In <i>node_modules</i> , you'll find the installed dependencies of your application.

Table 21.1 Files and Directories

File or Directory name	Purpose
<i>.gitignore</i>	In this file, you specify the files that shouldn't be committed to the repository.
<i>app.json</i>	This file contains the build configuration of the application.
<i>App.tsx</i>	The <i>App.tsx</i> file represents the entry point for the application.
<i>babel.config.js</i>	The <i>babel.config.js</i> file is used to configure the Expo presets.
<i>package-lock.json</i>	The <i>package-lock.json</i> file is used to fix the dependencies and their dependencies.
<i>package.json</i>	The <i>package.json</i> file allows you to describe your application as usual, such as with dependencies and scripts.
<i>tsconfig.json</i>	You use the <i>tsconfig.json</i> file to configure the TypeScript compiler for your application.

Table 21.1 Files and Directories (Cont.)

21.2.2 Starting the Application

In the default configuration, Expo provides a set of NPM scripts:

- **start**
Opens Expo Dev Tools in the browser. By default, the web interface is bound to port 19002.
- **android**
Opens Expo Dev Tools as for the start script, and also starts the application in the Android emulator.
- **ios**
Launches Expo Dev Tools and the iOS simulator.
- **web**
Again, launches Expo Dev Tools, and at the same time the application is opened in the browser.
- **eject**
As with Create React App, the `eject` script is used to export the configuration.

If you run the `npm start` command in the command line in your application, *Expo Terminal UI* will star. Along with your development environment, it represents the second central tool for developing your React Native application. [Figure 21.1](#) shows a screenshot of Expo Terminal UI after launch.

As you can see in the screenshot, the *Metro* bundler is used here. This is a module bundler, similar to Webpack. The difference is that Metro specializes in React Native and used to be a tightly integrated part of React Native as well. The task of the Metro bundler is to create a single file from an entry point and a set of source files.

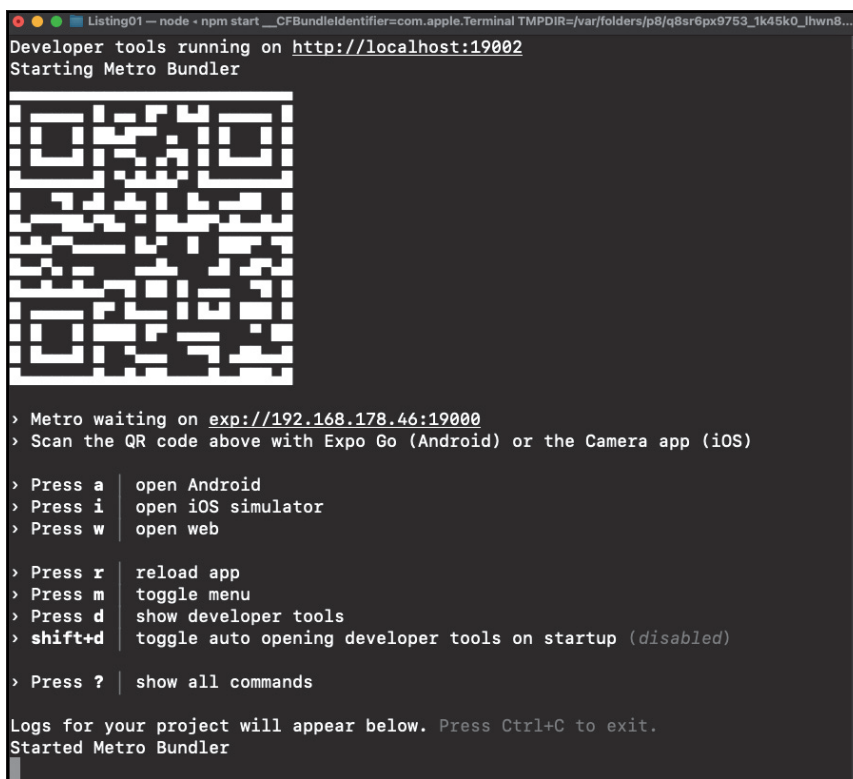


Figure 21.1 Expo Terminal UI View

At this point, you have a few options available:

- **Open the app in Android** (**A** key)
Opens the application in the Android emulator.
- **Open the app in iOS** (**I** key)
Opens the application in the iOS simulator.
- **Open the app on the web** (**W** key)
Opens the application in the browser.
- **Reload app** (**R** key)
Reloads the application in the simulator.
- **Display menu** (**M** key)
Displays the Expo menu in the simulator.
- **Display developer tools** (**D** key)
Opens the browser with the (deprecated) Expo Developer Tools.

With Expo, you can run your application on either a physical device or a simulator. We'll briefly introduce both variants in the following sections.

Running the Application in the Simulator

To run your application in a simulated environment, you need to make some preparations. The Expo team has documented these step by step. For the iOS simulator, you can find the instructions at <https://docs.expo.dev/workflow/ios-simulator/>, and for the Android studio emulator they're at <https://docs.expo.dev/workflow/android-studio-emulator/>.

Once you've set up the environment correctly, you can select your desired system in the Expo command line and press the `i` key for iOS, for example. This will launch the iOS simulator and open the Expo Go app, which will then display your application, and you can interact with it. [Figure 21.2](#) shows the current state of the application in the iOS simulator.

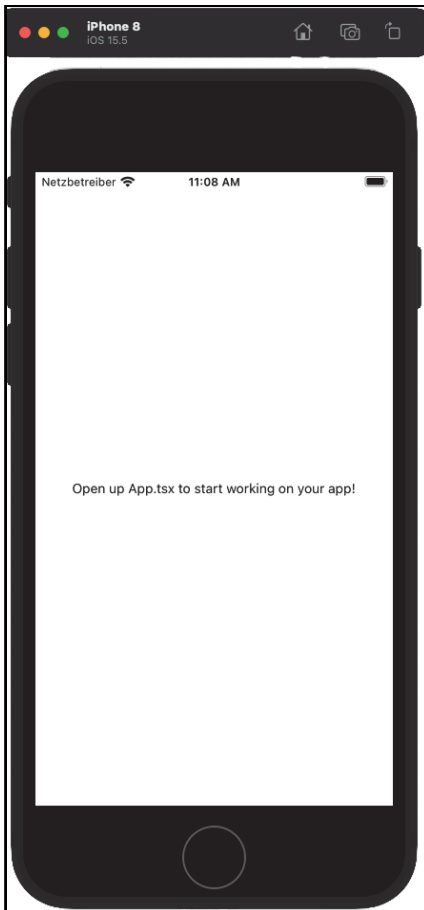


Figure 21.2 Application in iOS Simulator

Running the Application on a Device

To run your app on a real terminal device, you must first install the Expo Go app and create a free account. Then you can scan the QR code in the Expo command line and

open the link with Expo Go. This will open your application in Expo Go on your smartphone and you can start using your application.

Based on this initial application structure, you will implement an application in the following sections of this chapter. Let's start with the representation of a simple list.

21.3 Displaying an Overview List

For the most part, React Native follows the same implementation rules as React in the browser. The main differences lie in the fact that you cannot use HTML elements because React Native doesn't render HTML, but native GUI elements. Also, your application isn't delivered through a local web server. This means you need to open your app with a tool like Expo on a device, in a simulator, or in the browser. In the following sections, we'll use the iOS simulator.

21.3.1 Static List View

Typically, you divide your React Native app into multiple views between which you can navigate. For the initial list display, you create a new file named *List.tsx*. You need to place this file in the new *books/components* directory structure that needs to be created. The source code of this component is shown in [Listing 21.1](#):

```
import React from 'react';
import { FlatList, Text, View } from 'react-native';
import { Book } from '../Book';

const books = [
  {
    id: 1,
    title: 'JavaScript - The Comprehensive Guide',
    author: 'Philip Ackermann',
    isbn: '978-3836286299',
    rating: 5,
  },
  {
    id: 2,
    title: 'Clean Code',
    author: 'Robert Martin',
    isbn: '978-0132350884',
    rating: 2
  },
  {
    id: 3
    title: 'Design Patterns',
    author: 'Erich Gamma',
```

```

    isbn: '978-0201633610',
    rating: 5,
  },
];

const List: React.FC = () => {
  return (
    <View>
      <Text>Books management</Text>
      <FlatList<Book>
        data={books}
        keyExtractor={({item}) => item.id.toString()}
        renderItem={({ item }) => <Text>{item.title}</Text>}
      ></FlatList>
    </View>
  );
};

export default List;

```

Listing 21.1 List Display in React Native (books/components/List.tsx)

The element names from this example are fundamentally different from the ones you know from React in the browser. No `div` elements are used as containers; instead you use `View` components. Also, when displaying plain text, you must keep in mind that it cannot be placed anywhere in the app: texts must be rendered via the `Text` component. If you don't follow this rule, your application won't run and you'll receive a corresponding error message, as shown in [Figure 21.3](#).

This particular error occurs when you don't put the text of the heading—Books Management—between `Text` tags. To get the application back to its working state, you must correct the error and save the source code. The Expo process running in the background ensures that the view is automatically reloaded in the simulator.

For you to see the error at all, you still need to integrate the `List` component into the `App` component of your application. The source code required for this is shown in [Listing 21.2](#):

```

import React from 'react';
import List from '../books/components/List';

const App: React.FC = () => {
  return <List />;
};

export default App;

```

Listing 21.2 Integration of the “List” Component into the Application (App.tsx)

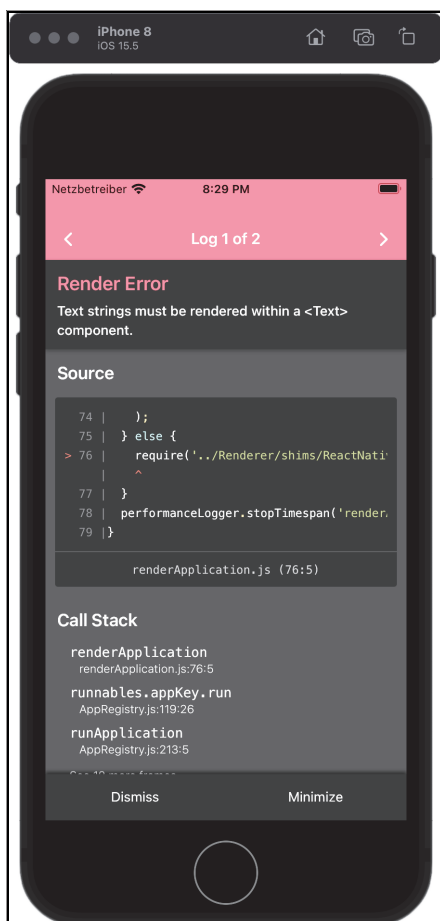


Figure 21.3 Error Messages in React Native

In the example, the list is displayed using the `FlatList` component (see [Listing 21.1](#)). The example is based on TypeScript, which allows you to specify types. The `FlatList` component is implemented as a generic component to which you can also pass a type in JSX. The `List` component is used to render an array of `Book` records. For the type specification, you define a new TypeScript type and place it in the `Book.ts` file within the `books` directory. The source code of this file is shown in [Listing 21.3](#):

```
export type Book = {
  id: number;
  title: string;
  author: string;
  isbn: string;
  rating: number;
};
```

Listing 21.3 “Book” Type (`books/Book.ts`)

For the records you currently still have in the `List` component to be displayed correctly, you must pass at least one `data` prop with the data array and one `renderItem` prop with a `render` function for the individual list items to the `FlatList` component. The `renderItem` function receives a `ListRenderItem` object as argument, in whose `item` property you'll find the data record to be rendered. To display the data records, you use the title of the `Book` object in a `Text` component.

If you implement this minimal version with the `data` and `renderItem` props, you'll receive an error message informing you of the missing `key` prop. The `key` prop is used by React Native in the same way that React uses it in the browser—namely, to assign each element to an iteration and reuse it in a re-render. You get the `key` prop by defining the `keyExtractor` prop and implementing a function that produces a unique string. In the example, the `id` property of the data record is used.

21.3.2 Styling in React Native

If you switch to the simulator with this state of the application, you'll still currently see a rather unspectacular view, as in [Figure 21.4](#).



Figure 21.4 Unstyled List View

Also, because React Native renders to a native environment rather than HTML and for the browser, you can't use the full feature set of CSS. However, React Native supports the object-oriented CSS syntax that you know from inline styles and numerous properties with their associated values.

Styling via the “StyleSheet” Object of React Native

React Native provides the `StyleSheet` object for styling the interface, whose `create` method lets you define a style object that you can in turn pass to the components of your app using the `style` prop. In [Listing 21.4](#), you can see the style definition for the list view. You can either integrate such styles directly into the component, which quickly becomes confusing with extensive stylesheets, or you can create a separate file for the styles. For the example, you create a new `List.styles.ts` file in the `admin/components/List` directory and paste the source code from [Listing 21.4](#) there.

```
import { StyleSheet } from 'react-native';

const styles = StyleSheet.create({
  headline: {
    marginTop: 30,
    fontSize: 20,
    fontWeight: 'bold',
    textAlign: 'center',
  },
  list: {
    marginTop: 20,
  },
  separator: {
    height: 1,
    borderTopWidth: 1,
    borderTopColor: 'darkgrey',
  },
  listItem: {
    marginHorizontal: 10,
    marginVertical: 20,
    display: 'flex',
    flexDirection: 'row',
    justifyContent: 'space-between',
  },
});

export default styles;
```

Listing 21.4 Styling of the List (`books/components/List.styles.ts`)

The stylesheet contains styles for the heading, the list itself, the individual entries, and the separators between the entries. With the exception of the property names, most declarations match standard CSS. However, there are also some deviations, such as when you specify the outer distance with the `margin` property: There, you can't specify multiple values and have to use properties like `verticalMargin` and `horizontalMargin`. The resulting `styles` object has the properties you passed when you created it. You then reference these properties in the `style` props of the components. [Listing 21.5](#) shows the integration of the styles into the `List` component:

```
import React from 'react';
import { FlatList, Text, View } from 'react-native';
import { Book } from '../Book';
import styles from './List.styles';

const books = [...];

const List: React.FC = () => {
  return (
    <View>
      <Text style={styles.headline}>Books management</Text>
      <FlatList<Book>
        style={styles.list}
        ItemSeparatorComponent={() => <View style={styles.separator} />}
        data={books}
        keyExtractor={(item) => item.id.toString()}
        renderItem={({ item }) => (
          <View style={styles.listItem}>
            <Text>{item.title}</Text>
            <Text>&gt;</Text>
          </View>
        )}
      ></FlatList>
    </View>
  );
};

export default List;
```

Listing 21.5 Applying the Styles in the “List” Component (books/components/List.tsx)

The previously defined styles are applied in the heading and the `FlatList` component. You also define a render function with the `ItemSeparatorComponent`, which you use to define a separator element that is rendered between the rows of the list. No separator is rendered before the first and after the last line. The separator element also gets a separate styling.

You can enhance the display of the individual entries by integrating the text into a `View` component that has a *flex layout*. This layout allows you to arrange the elements in a container very flexibly. The flex layout allows a frontend to be *responsive*—that is, to adapt dynamically to different screen sizes. Next to the text, you display an arrow icon to suggest an interaction option to the user. You can implement this option at a later time. With these adjustments, your application already looks a bit better, as you can see in [Figure 21.5](#).

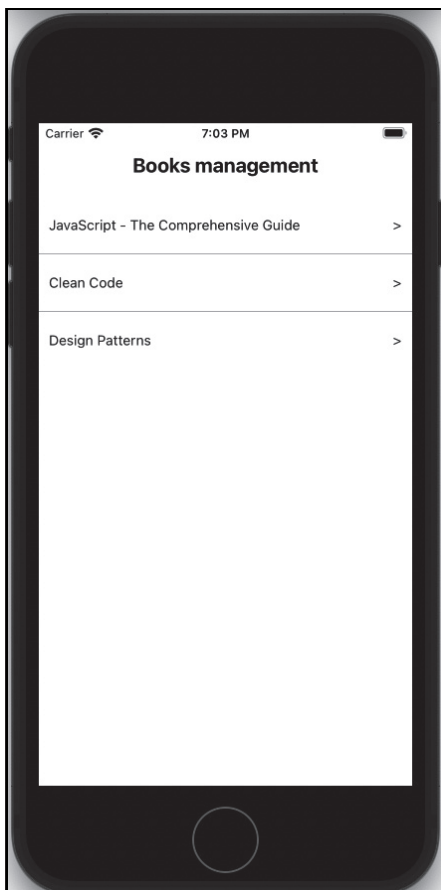


Figure 21.5 List View with Styles

As an alternative to `StyleSheet.create` in React Native, you can use the popular Emotion library, which you learned about in [Chapter 8](#).

Emotion in React Native

You can use the Emotion library in React Native for the purpose of abstraction as it allows you to use familiar CSS syntax. Emotion then translates the styles to the target environment. Instead of the React elements you used to use in the browser, here you

use components like `View` or `Text` offered to you by React Native. Before you can use Emotion in your application, you must install the base package as well as the extension for React Native using the `npm install @emotion/react @emotion/native` command. [Listing 21.6](#) contains the `List.styles.ts` file customized for Emotion:

```
import styled from '@emotion/native';

export const Headline = styled.Text`
  margin-top: 30px;
  font-size: 20px;
  font-weight: bold;
  text-align: center;
`;

export const Separator = styled.View`
  height: 1px;
  border-top-width: 1px;
  border-top-color: darkgrey;
`;

export const FlatList = styled.FlatList`
  margin-top: 20px;
`;

export const ListItem = styled.View`
  margin: 20px 10px;
  display: flex;
  flex-direction: row;
  justify-content: space-between;
`;
```

Listing 21.6 Emotion Styling for the List View (books/components/List.styles.ts)

By using Emotion, you get two benefits in your application: you swap out the use of the style props from within the component, which makes it look tidier, and you can assign meaningful names to the components styled with Emotion, making the JSX structure more self-explanatory. You can see the result in the source code of the `List` component shown in [Listing 21.7](#):

```
import React from 'react';
import { Text, View } from 'react-native';
import { Book } from '../Book';
import { Headline, Separator, FlatList, ListItem } from '../List.styles';

const books = [...];
```

```
const List: React.FC = () => {
  return (
    <View>
      <Headline>Books management</Headline>
      <FlatList
        ItemSeparatorComponent={() => <Separator />}
        data={books}
        keyExtractor={(item) => (item as Book).id.toString()}
        renderItem={({ item }) => (
          <ListItem>
            <Text>{(item as Book).title}</Text>
            <Text>&gt;</Text>
          </ListItem>
        )}
      ></FlatList>
    </View>
  );
};

export default List;
```

`export default List;`

Listing 21.7 “List” Component with Emotion Components (books/components/List.tsx)

If the list has a larger number of entries, it will quickly become confusing for your users. In such a case, you can provide a search field that can be used to reduce the number of elements in the list.

21.3.3 Search Field for the “List” Component

A combination of familiar React patterns is used for the search field in the `List` component. The input field you set up with the `TextInput` component is implemented as a controlled component. To capture the local state of the filter, you use the state hook of the React Hook API. You use this value to filter the data to be displayed. By default, the input field has no frame, so it’s hard to find. For this reason, you draw a frame around the input field using an Emotion component. To do so, you must first extend the *List.styles.ts* file as shown in [Listing 21.8](#):

```
...
export const Search = styled.TextInput`
  border: 1px solid darkgrey;
  padding: 10px 5px;
  margin-top: 10px;
`;
```

Listing 21.8 Extension of the “List” Component Styles (books/components/ List.styles.ts)

In [Listing 21.9](#), you can see the corresponding adjustment:

```
import React, { useState } from 'react';
import { Text, View } from 'react-native';
import { Book } from '../Book';
import {
  Headline, Separator, FlatList, ListItem, Search
} from './List.styles';

const books = [...];

const List: React.FC = () => {
  const [filter, setFilter] = useState('');
  return (
    <View>
      <Headline>Books management</Headline>
      <Search
        autoCapitalize="none"
        value={filter}
        onChangeText={(text: string) => setFilter(text)}
        placeholder="Search"
      />

      <FlatList
        ItemSeparatorComponent={() => <Separator />}
        data={books.filter((book) =>
          book.title.toLowerCase().includes(filter.toLowerCase())
        )}
        keyExtractor={(item) => (item as Book).id.toString()}
        renderItem={({ item }) => (
          <ListItem>
            <Text>{(item as Book).title}</Text>
            <Text>&gt;</Text>
          </ListItem>
        )}
      ></FlatList>
    </View>
  );
};

export default List;
```

Listing 21.9 Search Function for the “List” Component (books/components/List.tsx)

You initialize the `filter` state of the component with an empty string. In the styled `TextInput` component, which is the `Search` component, you use the `placeholder` prop to indicate that the form is a search field. In addition, you use the `autoCapitalize` prop with the value `none` to prevent the system from trying to automatically capitalize the first letter of the entered text. The `value` and `onTextChanged` props must be used to turn the `TextInput` component into a controlled component and link it to the state.

Currently, the data to be displayed is still located directly in the component. The next step is to add to enable your app to communicate with a server in order to load the data.

21.3.4 Server Communication

The data you use in a React Native app rarely stays within the app but is instead read from and written to a server. For this purpose, React Native implements the `Fetch` API of the browser. In addition, the `XMLHttpRequest` API is supported, so a wide range of third-party libraries are also supported, such as `Axios`. To load the data from the server, you need a combination of a state hook in which you hold the data, an effect hook that you use when initializing the component to load the data from the server, and the `Fetch` API that actually implements this loading process. When it comes to implementing this feature, React in the browser and React Native are no different, as you can see in [Listing 21.10](#):

```
import React, { useEffect, useState } from 'react';
import { Text, View } from 'react-native';
import { Book } from '../Book';
import { Headline, Separator, FlatList, ListItem, Search } from '../List.styles';

const List: React.FC = () => {
  const [filter, setFilter] = useState('');
  const [books, setBooks] = useState<Book[]>([]);

  useEffect(() => {
    fetch('http://localhost:3001/books')
      .then((response) => response.json())
      .then((data) => setBooks(data));
  }, []);

  return (
    <View>
      <Headline>Books management</Headline>
      <Search
        autoCapitalize="none"
        value={filter}
        onChangeText={(text: string) => setFilter(text)}
      />
    </View>
  );
};
```

```

        placeholder="Search"
      />

      <FlatList
        ItemSeparatorComponent={() => <Separator />}
        data={books.filter((book) =>
          book.title.toLowerCase().includes(filter.toLowerCase())
        )}
        keyExtractor={(item) => (item as Book).id.toString()}
        renderItem={({ item }) => (
          <ListItem>
            <Text>{(item as Book).title}</Text>
            <Text>&gt;</Text>
          </ListItem>
        )}
      ></FlatList>
    </View>
  );
};

export default List;

```

Listing 21.10 Loading the Initial Data from the Server (books/components/List.tsx)

For the server communication to work, you install the `json-server` package in your application using the `npm install json-server` command and create a file named `data.json` in the root directory. You can see the contents of this file in [Listing 21.11](#):

```

{
  "books": [
    {
      "id": 1,
      "title": "JavaScript - The Comprehensive Guide",
      "author": "Philip Ackermann",
      "isbn": "978-3836286299",
      "rating": 5
    },
    {
      "id": 2,
      "title": "Clean Code",
      "author": "Robert Martin",
      "isbn": "978-0132350884",
      "rating": 4
    },
    {

```

```
    "id": 3
    "title": "Design Patterns",
    "author": "Erich Gamma",
    "isbn": "978-0201633610",
    "rating": 5
  }
]
```

Listing 21.11 Data for the Backend (data.json)

You start the server in the command line using the `npx json-server -p 3001 -w data.json` command. Alternatively, you can place this command as an NPM script in your `package.json` file. When you switch to the simulator now, you can reload your app by pressing the `[R]` key, and you'll then see the data records from the server.

While developing a native app with React Native, you don't have to sacrifice the convenience of sophisticated development tools for debugging and analysis, despite the simulated environment.

21.4 Debugging in the Simulated React Native Environment

You can access the debugging menu via a keyboard shortcut, which varies depending on the environment. Table 21.2 summarizes them for you.

Simulator/Operating System	Windows	macOS
Android	<code>[Ctrl] + [M]</code>	<code>[Command] + [M]</code>
iOS		<code>[Command] + [D]</code>

Table 21.2 Keyboard Shortcuts to Open the Debugging Menu

There are a number of options available in the debugging menu of the simulator (see Figure 21.6). One of the most important to consider is **Debug Remote JS**.

If you enable this option, a new tab will open in your locally installed Chrome browser, pointing to the following URL: `http://localhost:19001/debugger-ui/`. The JavaScript process of the React Native application is executed in a separate worker process. You can reach this by opening the developer tools of the browser, switching to the **Sources** tab, and selecting the entry there that starts with `debuggerWorker`. Here you can find the files and directories of your application, set breakpoints, and analyze the runtime environment. In Figure 21.7, you can see an activated breakpoint in the `List` component of the app in the debugger.

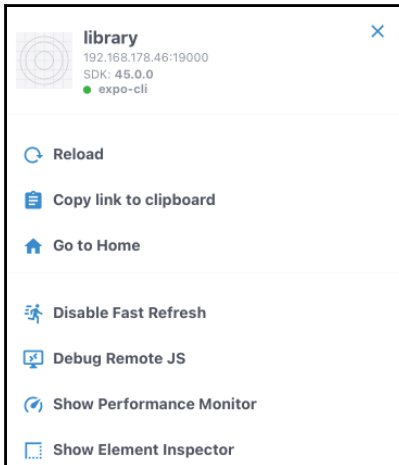


Figure 21.6 Debugging Menu in the iOS Simulator

In addition to Chrome Developer Tools, you can install the *standalone version* of React Dev Tools, an Electron application that allows you to examine the structure and layout of your app. *Electron* is a platform that allows you to implement desktop applications with JavaScript.

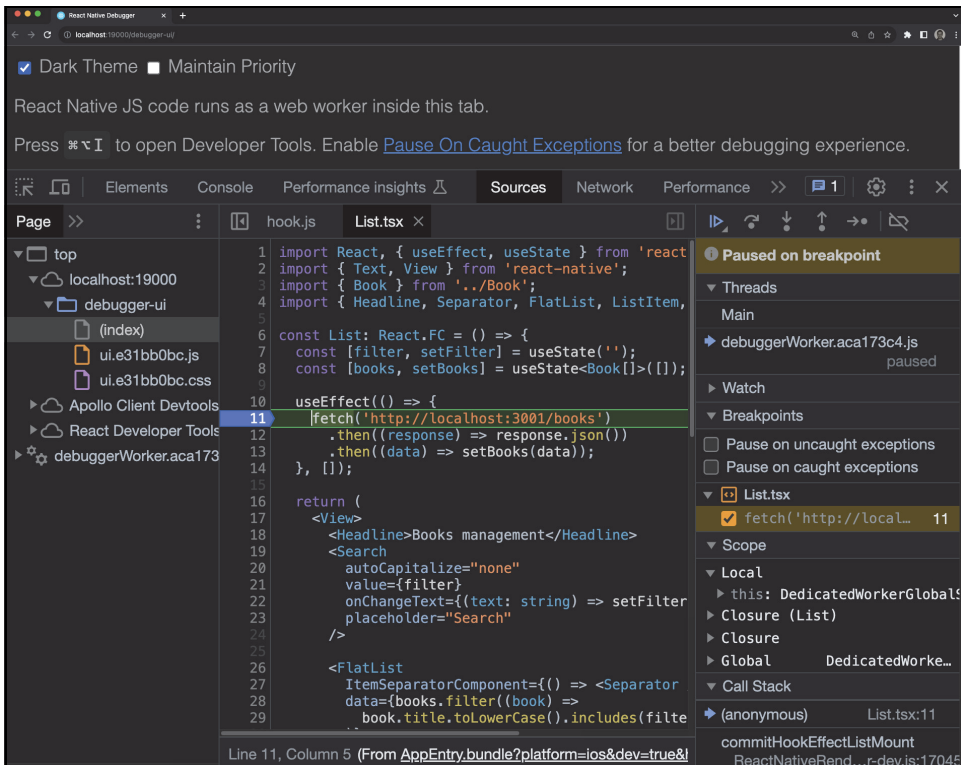


Figure 21.7 Debugging in the iOS Simulator Environment

React Dev Tools can be installed using the `npm install -g react-devtools` command. With a subsequent call of the `react-devtools` command in the command line, you can open the tool. React Dev Tools automatically connects to a React Native application that's running in debugging mode on the local system. In [Figure 21.8](#), you can see the React Dev Tools view with the current state of the application.

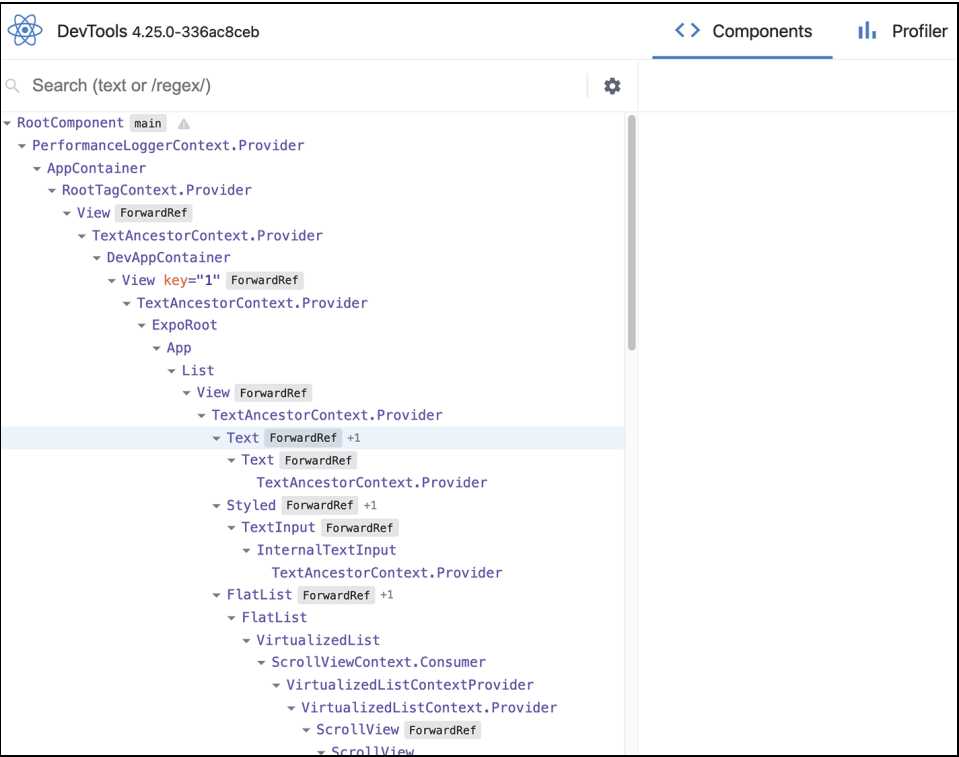


Figure 21.8 Application in React Dev Tools

After displaying the data records, the next step is to enable editing the records in the application.

21.5 Editing Data Records

Usually, a React Native app doesn't consist of only one view. As in the browser, you need an additional package in React Native to help you manage the transitions between the different views of your application. For this purpose, you can use various implementations such as React Navigation or React Router. For the sample app, we use the native version of React Router and deploy it to navigate from the list view to the form for editing an item and then back again.

You can install React Router Native via the `npm install react-router-native` command. The package already contains the appropriate type definitions, so you don't need to install anything else. The integration of the router is similar to that in the browser. In the `App` component, you first integrate the `NativeRouter` component and then the `Routes` component directly within it. Inside that component, you can define your individual routes using the `Route` component. The source code of the extended `App.tsx` file is shown in [Listing 21.12](#):

```
import React from 'react';
import { NativeRouter, Routes, Route, Navigate } from 'react-router-native';

import List from './books/components/List';
import Form from './books/components/Form';

const App: React.FC = () => {
  return (
    <NativeRouter>
      <Routes>
        <Route path="/list" element={<List />} />
        <Route path="/edit/:id" element={<Form />} />
        <Route path="/" element={<Navigate to="/list" />} />
      </Routes>
    </NativeRouter>
  );
};

export default App;
```

Listing 21.12 Routing in the React Native App (`App.tsx`)

In this case, you define two routes for your app. The `/list` path takes you to the list view, and `/edit` followed by the ID of the data record you want to edit renders the `Form` component that lets you modify the data record. The default route users take to get into the app makes sure that the app redirects to the `/list` path. The `Form` component is responsible for displaying the form and takes care of storing the data. For the form, you first use a placeholder component that you save in the `books/components` directory in the `Form.tsx` file. The source code of this component is shown in [Listing 21.13](#):

```
import React from 'react';
import { Text, View } from 'react-native';

const Form: React.FC = () => {
  return (
    <View>
      <Text>Form works!</Text>
    </View>
  );
};
```

```
    </View>
  );
};

export default Form;
```

Listing 21.13 Placeholder Component for the “Form” Component

This status of the `Form` component is only temporary. We’ll deal with the final implementation in the next section.

21.5.1 Implementing the “Form” Component

As you have already seen in the `List` component, React Native provides input elements. These provide the basic functionality you’re used to in the browser environment. However, this standard component comes with no or very little styling and few additional features.

A solution for this is provided by additional libraries such as `react-native-elements`. You can install this library using the `npm install react-native-elements` command. `React-native-elements` brings its own type definitions, so you don’t need to install any additional packages in this context. The `react-native-elements` package has a peerDependency on the `react-native-vector-icons` package. For the Expo environment you’re currently developing in, you install the `@expo/vector-icons` package via the `npm install @expo/vector-icons` command instead of this package. After that, you can turn your attention to the implementation of the `Form` component. You can get information about which record is being processed via the route. Using this information, you load the data from the server and store it in a local state. For each property of the data record you want to edit, you insert an input field. To allow the user to submit the form, you also add a `Button` component. When you press the button, the data is sent back to the server, after which the `List` component is displayed again. In [Listing 21.14](#), you can see the source code of the `Form` component:

```
import React, { useCallback, useEffect, useState } from 'react';
import { View } from 'react-native';
import { useNavigate, useParams } from 'react-router-native';
import { Book } from '../Book';
import { Headline } from './List.styles';
import { Input, Button } from 'react-native-elements';

const Form: React.FC = () => {
  const width = 250;
  const marginTop = 20;
  const marginLeft = 10;
```

```

const navigate = useNavigate();
const { id } = useParams<{ id: string }>();
const [book, setBook] = useState<Omit<Book, 'rating'>>({
  id: 0,
  title: '',
  author: '',
  isbn: '',
});

const goBack = useCallback(() => navigate('/list'), []);

useEffect(() => {
  fetch(`http://localhost:3001/books/${id}`)
    .then((response) => response.json())
    .then((data) => setBook(data));
}, [id]);

async function save() {
  await fetch(`http://localhost:3001/books/${book.id}`, {
    method: 'PUT',
    headers: {
      'Content-type': 'application/json',
    },
    body: JSON.stringify(book),
  });
  goBack();
}

const createChangeHandler = (name: string) => (text: string) =>
  setBook((prevBook) => ({
    ...prevBook,
    [name]: name !== name ? parseFloat(text) : text,
  }));

return (
  <View style={{ position: 'absolute', top: 0 }}>
    <Headline>{book.title} edit</Headline>
    <Input
      label="Title"
      placeholder="Title"
      value={book.title}
      containerStyle={{ width, marginTop, marginLeft }}
      onChangeText={createChangeHandler('title')}
      autoCompleteType="off"
    />
  />

```



```
<Input
  label="Author"
  placeholder="Author"
  value={book.author}
  containerStyle={{ width, marginTop, marginLeft }}
  onChangeText={createChangeHandler('author')}
  autoCompleteType="off"
/>
<Input
  label="ISBN"
  placeholder="ISBN"
  value={book.isbn}
  containerStyle={{ width, marginTop, marginLeft }}
  onChangeText={createChangeHandler('isbn')}
  autoCompleteType="off"
/>
<Button
  onPress={save}
  title="Save"
  buttonStyle={{ marginTop, marginLeft: marginLeft + 10, ←
    width: 230 }}
/>
<Button
  onPress={goBack}
  title="Cancel"
  type="outline"
  buttonStyle={{ marginTop: 5, marginLeft: marginLeft + 10, ←
    width: 230 }}
/>
</View>
);
};

export default Form;
```

Listing 21.14 Form for Editing Data Records (books/components/Form.tsx)

Initially, the local state of the `Form` component contains a `Book` object with empty values. These are filled with concrete values by calling the `Fetch` function within the effect hook. The server provides a `/book/:id` route that you can use to load a single record. For the `id` placeholder, you use the ID that was passed via the frontend route. Via the `useParams` hook, you get the corresponding value.

The `save` function works similarly, but with the difference that it uses the `PUT` method to send the modified data record to the server as a JSON structure. The `goBack` helper function navigates back to the list after the save operation. This function is also

stored as a press handler for the **Cancel** button. To make the navigation work in the Form component, you use the `useNavigate` hook of React Native Router, which provides the `navigate` function.

The individual input elements are implemented as controlled components and as such are linked to the state of the component. The `label` and `placeholder` props tell users what value the field represents. The `containerStyle` prop enables you to influence the spacing and dimensions of the form fields. The same applies to the **Save** and **Cancel** buttons: again, you define an event handler that's activated when the button is pressed. The `title` prop determines the label of the button, while the `buttonStyle` prop affects its appearance.

The `react-native-elements` package provides for different button styles. The default style consists of a filled button. In addition, you can use the `type` prop to select the outline style, for example. This means that the button isn't filled and only has a frame. When you use this style, the **Cancel** button will move somewhat into the background.

To make sure you can access the form, you add the `Link` component of React Router to the `List` component. [Listing 21.15](#) shows how this works:

```
import React, { useEffect, useState } from 'react';
import { Text, View } from 'react-native';
import { Link } from 'react-router-native';
import { Book } from '../Book';
import { Headline, Separator, FlatList, ListItem, Search } from ←
  './List.styles';

const List: React.FC = () => {
  const [filter, setFilter] = useState('');
  const [books, setBooks] = useState<Book[]>([]);

  useEffect(() => {...}, []);

  return (
    <View>
      <Headline>Books management</Headline>
      <Search ... />

      <FlatList
        ItemSeparatorComponent={() => <Separator />}
        data={books.filter((book) =>
          book.title.toLowerCase().includes(filter.toLowerCase())
        )}
        keyExtractor={(item) => (item as Book).id.toString()}
        renderItem={({ item }) => (
          <ListItem>
```

```

        <Text>{(item as Book).title}</Text>
        <Link to={` /edit/${(item as Book).id}`}>
          <Text>&gt;</Text>
        </Link>
      </ListItem>
    )}
  ></FlatList>
</View>
);
};

```

```
export default List;
```

Listing 21.15 Integration of the “Edit” Link in the List (books/components/List.tsx)

If you save this state of the app and switch to the simulator, you can click or tap on a list entry to reach the form where you can modify the individual values. [Figure 21.9](#) shows what the form looks like in the iOS simulator.

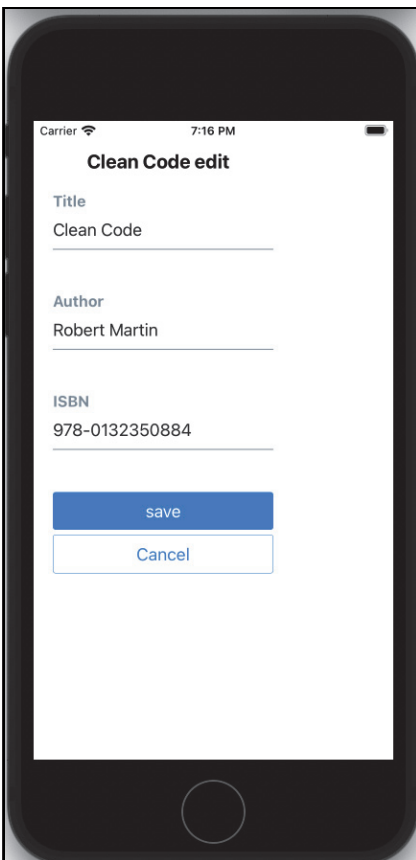


Figure 21.9 Form View

At this stage, the application is far from being fully implemented, but you should've gotten an idea of the differences and similarities between React and React Native. In the next step, we'll address one of the biggest differences between the two platforms—namely, in publishing a React Native app.

21.6 Publishing

Once you've completed the feature implementation of your app and are content with the result, one last step remains: publishing your app to Apple's App Store or the Google Play Store. Expo will support you in this respect as well. Before you can upload your app to a store, you must build the app. The essential aspects of the build are specified via the *app.json* file. Here you should pay particular attention to the fact that the name, icon, and version information is maintained. An example of the *app.json* file is shown in [Listing 21.16](#):

```
{
  "expo": {
    "name": "library",
    "slug": "library",
    "version": "1.0.0",
    "orientation": "portrait",
    "icon": "./assets/icon.png",
    "userInterfaceStyle": "light",
    "splash": {
      "image": "./assets/splash.png",
      "resizeMode": "contain",
      "backgroundColor": "#ffffff"
    },
    "updates": {
      "fallbackToCacheTimeout": 0
    },
    "assetBundlePatterns": [
      "**/*"
    ],
    "ios": {
      "supportsTablet": true
    },
    "android": {
      "adaptiveIcon": {
        "foregroundImage": "./assets/adaptive-icon.png",
        "backgroundColor": "#FFFFFF"
      }
    }
  },
}
```

```
"web": {  
  "favicon": "../assets/favicon.png"  
}  
}  
}
```

Listing 21.16 app.json File of the Application

After checking the *app.json* file of your application, you should make sure that the Expo CLI is installed on your system. If the `expo` command isn't available on the command line, you should run the `npm install -g add expo-cli` command.

Once you've met all the requirements, you can build your application. Note that you need a free Expo account for the build. You can generate the application either on the Expo website or directly from the console. Then you can run the build from the command line. For an Android build, you need to run the `expo build:android` command, and for an iOS build, run `expo build:ios`. In the case of Android, you can choose between an APK and an app bundle build, the latter being the recommended variant. You can create the app bundle via the `expo build:android -t app-bundle` command. For more information and step-by-step instructions on how to build your application, visit <https://docs.expo.dev/classic/building-standalone-apps>. Once you've created the build, you can upload your app to Apple's App Store or the Google Play Store. For more information on this, visit <https://docs.expo.dev/distribution/uploading-apps/>.

21.7 Summary

In this chapter, you learned about *React Native*, an additional renderer for React that helps you run your React app natively on a mobile device:

- React Native renders the graphical user interface in a native environment, which is written in Objective-C or Swift for iOS and in Java or Kotlin for Android. The app's JavaScript logic runs in a separate JavaScript environment.
- You can use Expo as a tool for numerous tasks as part of developing an app using React Native. These range from initialization to execution in the simulator to publishing the app.
- You also learned how a React Native app is built and structured.
- Unlike in the browser, you cannot use CSS in React Native. However, the environment provides a subset of the properties and values known from CSS.
- By using controlled components, you can create form constructs similar to those in the browser and synchronize the values of a form with the local state of a component.

- To communicate with a server, React Native provides the Fetch API, which you know from the browser environment. Libraries such as Axios that are built on this interface, and the XMLHttpRequest object, also work in React Native.
- Using the debugging tools provided by React Native, you can inspect the elements of the user interface and use the JavaScript debugger in its full functionality as usual.
- Once you've finalized the functionality of your app, you can build your app and publish it using Expo.

You've now learned a lot about React and its ecosystem. You know what the library can do and, even more importantly, can evaluate and rank architecture and design patterns as well as libraries, extensions, and even new features. This means that you have the tools not only to deal with implementing your own React application, but also to dig into and solve future problems.

We wish you a lot of fun and success on your journey with React!

The Author



Sebastian Springer is a JavaScript engineer at MaibornWolff. In addition to developing and designing both client-side and server-side JavaScript applications, his focus is on imparting knowledge. As a lecturer for JavaScript, a speaker at numerous conferences, and an author, he inspires enthusiasm for professional development with JavaScript. Sebastian was previously a team leader at Mayflower GmbH, a premier web development agency in Germany. He was responsible for project and team management, architecture, and customer care for companies such as Nintendo of Europe, Siemens, and others.

Index

.flowconfig 221

A

Action creator 430
Action object 192
 payload 192
 type 192
Airbnb style guide 82
Alternatives 33
Angular 28
Anonymous function 233
Apollo 514
 @apollo/react-hooks 515
 apollo-boost 515
 ApolloClient 516
 ApolloProvider 516
 authentication 532
 Authorization header 532
 Chrome extension 526
 client 509
 Context 515
 data 518
 deleting 522
 error 518
 generic functions 520
 graphql 515
 initialization 527
 InMemoryCache 516
 installation 515
 list representation 516
 loading 518
 loading indicator 518
 local-only fields 527
 local resolver 527
 local state 528
 makeVar 528
 mutation hook 522
 reactive variables 527
 read 516
 refetchQueries 522
 state management 527
 states 518
 useReactiveVar 530
Apollo Client Devtools 526
 cache 526
 explorer 526
 mutations 526
 queries 526

App.jsx 77
AppBar 387
Application logic 137
aria 366
Arrow function 82
Asset, static 67
async/await 131
Asynchronous operation 130, 476
Autofocus 308
AWS
 configuration 136
Axios 135, 238, 332
 instance 136

B

Babel 47, 50, 54
 JSX 50
 plugin 258
 text/babel 50
Backend implementation 332
Barrier to entry 188
Basic hooks 188, 215
Basic principles 73
beforeinstallprompt 614
Bootstrap 347
Breaking change 29, 33, 60
Breakpoint 68
Browser cache 28
Browser extension 70
Browser features 605
BrowserRouter 384
 integration 385
Build
 JSX Transform 81
 translation 81
Building block 34
Build process 52, 71, 383, 408, 469
 directory 71
 web server 71
Bundler 82
By reference 580

C

CDN 50, 347, 620
Central state management 213
Central status management 455

- Change event 334
- Changelog 60
- Child component 42, 95, 115
- Child element 36
- children prop 147
- Chrome 53
- Class component 35, 161, 217
 - constructor* 167
 - context property* 185
 - data type* 163
 - defaultProps* 165
 - default values* 165
 - initializing the state* 166
 - props* 163
 - props object* 171
 - readability* 165
 - rendering* 162
 - state* 100, 166
 - state property* 166
 - static property* 164
- Classes versus functions
 - lifecycle* 186
 - state* 185
- Class name 248
 - conditional* 250
 - merging* 250
 - multiple* 250
- Cleanup 124
- clearInterval 124
- closest 109
- Code examples 42
- Codemod 34, 179
- CodePen 46
 - login* 47
 - React* 47
- Command line 631
- Command line tool 52
- Commit 170
- Commit phase 182
- Compiler 219
- Component 34, 73, 161
 - file extension* 80, 307
 - splitting* 139
 - state* 100
 - tag* 97
 - uppercase letters* 86
- Component collection 40
- componentDidCatch 179
 - logging* 180
- componentDidMount 173
- componentDidUpdate 175
- Component function 87
- Component hierarchy 41, 115
- Component library 347, 407
- Component lifecycle 169
- Component stack trace 180
- Component tree 35, 73, 158, 427
- componentWillMount 178
- componentWillReceiveProps 171, 179
- componentWillUnmount 177
- componentWillUpdate 179
- Composition 188
- Compressor 584
- Concurrent mode 31
- Confirmation dialog 365
- const 232
- Constructor 169, 171
- Container component 137
 - integration* 140
- Content delivery network -> see CDN 347
- Context 150, 427, 438
 - access* 153
 - child elements* 155
 - Consumer* 151
 - createContext* 151
 - creation* 151
 - example* 154
 - integration* 152
 - Provider* 151
- Context API 31, 183
- contextType 31
- Controlled component 305, 318
 - onChange handler* 318
 - synchronizing* 319
 - value prop* 318
- Cordova 631
- createElement 86
- Create React App 27, 45, 52, 54
 - installation* 54, 57
 - npm* 55
 - options* 58
 - template* 59, 226
 - TypeScript* 46
 - yarn create* 58
- createRoot 75
- cross-env 134
- Cross-origin resource sharing (CORS) 132
- CRUD 43
- CSS 83, 245
 - base selectors* 247
 - CamelCase* 255
 - class names* 248
 - global* 84
 - importing* 83

<i>namespacing</i>	246
<i>properties</i>	246
<i>root element</i>	246
<i>rules</i>	255
<i>selectors</i>	246
<i>specificity</i>	249
<i>unit</i>	255
CSS framework	266
CSS import	245
CSS-in-JS	257
CSS module	255
<i>class names</i>	257
<i>Create React App</i>	257
<i>file extension</i>	255
<i>naming scheme</i>	257
<i>pseudo selectors</i>	257
<i>standards-compliant</i>	256
CSS preprocessor	252
Custom hook	214, 325, 357, 615

D

Data class	87
Data flow	36, 42, 115
Data stream	36
Data tables	551
data-testid	295
Debugger	68
<i>navigation</i>	68
<i>Visual Studio Code</i>	70
<i>WebStorm</i>	70
Debugging	49
Default browser	63
Default export	82
Default port	127, 132
DefinitelyTyped	230
Dependency array	198
Deployment	132
Deprecated	178
Deprecation	34
describe.each	394
Design pattern	142, 218
Destructuring	48
devDependency	230
Developer tools	68
Development	
<i>local</i>	49
Development process	45, 52
Dexie	626
<i>installation</i>	626
Dialog box	364
Dialog control	373
Dialogue, open prop	365
Dispatch	192–193
DOM	37
Dumb component	137

E

ECMAScript	220
ECMAScript module system	62
Edge	53
Effect hook	116
Electron	649
Element	34, 85
<i>attributes</i>	85
<i>immutable</i>	86
Emotion	257
<i>autocompletion</i>	261
<i>css prop</i>	258
<i>css template string</i>	260
<i>external object</i>	260
<i>inline object</i>	260
<i>installation</i>	258
<i>styled approach</i>	258, 260
<i>syntax highlighting</i>	261
Encryption	66, 132
Entry file	67
Environment variables	133
<i>.env</i>	133
<i>command line</i>	133
<i>NODE_ENV</i>	133
<i>REACT_APP</i>	133
Epic	492
Error boundary	179
Error handling	179
error prop	313
ESLint	75, 78, 215
<i>error</i>	79
eslintrc.json	79
Event binding	102
Event handler	37, 106
Event handling	108
Event object	107
Event pooling	107
Events	102
<i>callback</i>	103
<i>click</i>	102
Exception	69
Expo	632
Expo CLI	632
Export	82
<i>named</i>	82
Expo Terminal UI	633

F

Fab	375
Facade	512
Facebook	27
FaxJS	29
Fetch API	128
Fetch then render	598
Fiber	30, 33, 187
File extension	74
File name	80
files property	330
File upload	328
<i>uncontrolled component</i>	328
fireEvent.change	312
Firefox	53
Flexbox	359
Flow	220
<i>@flow</i>	222
<i>base types</i>	224
<i>build process</i>	223
<i>command line</i>	223
<i>configuration</i>	221
<i>declaration</i>	222
<i>flow-remove-types</i>	221
<i>generics</i>	224
<i>integrating in React</i>	221
<i>integration</i>	221
<i>interfaces</i>	224
<i>props</i>	224
<i>React</i>	224
<i>state</i>	224
<i>Visual Studio Code</i>	222
<i>WebStorm</i>	222
Flow Language Support	222
Flux	40, 189, 427
<i>action</i>	429
<i>dispatcher</i>	430
<i>representation</i>	429
<i>separating specialties</i>	428
<i>store</i>	428
<i>view</i>	428
Flux standard action	429
<i>criteria</i>	429
<i>payload</i>	429
<i>type</i>	429
Form	305
<i>styling</i>	342
<i>submitting</i>	309
<i>test</i>	323
FormData	330
Form dialog	373, 402
Formik	305
Form validation	334
<i>testing</i>	343
ForwardRefs	202
Fragment	158
Function component	36, 78, 187

G

Generator	474
Generator function	474, 477
getDerivedStateFromError	179
<i>alternative representation</i>	182
getDerivedStateFromProps	171
getSnapshotBeforeUpdate	175
GitHub	27
Git repository	66
<i>installation</i>	418
Global stylesheet	246
Gradual upgrade	32
GraphQL	513, 526
<i>autocompletion</i>	514
<i>error highlighting</i>	514
GraphQL	509
<i>application logic</i>	514
<i>caching</i>	510
<i>data structure</i>	511
<i>data types</i>	512
<i>deprecated</i>	510
<i>disadvantages</i>	510
<i>documentation</i>	510
<i>filter</i>	513
<i>gql</i>	517
<i>graphs</i>	509
<i>input type</i>	514
<i>mutations</i>	514
<i>overhead</i>	510
<i>plain text</i>	510
<i>query</i>	513
<i>query language</i>	510
<i>resolver</i>	512
<i>schema</i>	511
<i>structure</i>	509
<i>type system</i>	510
<i>versioning</i>	510
GraphQL Code Generator	520
Grid system	358

H

HashRouter	384
Higher-order component	142, 581

1Personal Copy for Jaleel Hussain, alex76alex43@gmail.com 667

installation 273
interaction 297
it 276
jest.fn 298
matcher 277
mock backend 299
mockReturnValue 288
mocks 288
only 282
organization 274
promises 285
props 294
random numbers 288
reference snapshot 291
rejects 286
resolves 286
server communication 287
server dependency 299
server error 301
setup routines 279
side effect 287
skip 281
snapshot 41, 289, 293
spy function 298
state 294
teardown routines 279
test 276
test suites 279
timeout 285
toHaveBeenCalledWith 298
toMatchInlineSnapshot 294
toMatchSnapshot 294
toThrow 283
type checking 284
JSCodeshift 61
jsdom 273
JSON 130
json-server 126
JSON Web Token -> see JWT 498
JSX 34, 74, 76, 84, 248
 AND operator 92
 comment 89
 conditions 85, 91
 dangerouslySetInnerHTML 89
 exception 89
 expressions 86
 extensibility 89
 if statements 92
 iterations 89
 JavaScript expressions 85
 JavaScript functionality 90
 loops 84, 89

map 90
protection mechanism 89
static values 86
syntax extension 84, 86
Tternary operator 93
undefined 88
JWT 498
 access token 499
 validity 498

K

Karma 41
key attribute 91
key prop 38–39
 assignment 39

L

lazy (function) 31
Lazy loading 28
let 232
Library 407
 build 415
 build process 414
 central export 413
 component test 419
 directory structure 411
 dist directory 414
 entry point 413
 export 412
 hooks 413
 hook test 420
 initialization 408
 integrating 415
 main 415
 structure 411
 testing 419
 types 415
Library hooks 213
Lifecycle 36, 115
 mount 115–116
 mounting 116
 unmount 115, 123
 update 115, 119
 updating 119
Lifecycle hook 170
Lifecycle method 31
Lighthouse 628–629
Linux Foundation 509
Loader 82
Loading indicator 119

Loading process	555
Local development	49
Logic	137
Login form	308
LowDB	572
Low priority	211

M

Mangler	584
manifest.json	67
Material	347
Material Design	40
Material UI	40, 104, 347
<i>anchorEl</i>	389
<i>breakpoint</i>	358
<i>controlled component</i>	353
<i>creating</i>	369
<i>deleting</i>	363
<i>dialog</i>	364
<i>DialogActions</i>	366
<i>DialogContent</i>	366
<i>DialogTitle</i>	366
<i>editing</i>	376
<i>filters</i>	352
<i>floating action button</i>	375
<i>grid</i>	358
<i>icons</i>	348, 361
<i>installation</i>	347
<i>interaction</i>	361
<i>keepMounted</i>	389
<i>scrollbar</i>	360
<i>sorting</i>	355
<i>sorting indicators</i>	357
<i>spacing</i>	359
<i>sx</i>	389
<i>table</i>	348
<i>TableSortLabel</i>	355
<i>TextField</i>	370
Memo hook	198
Memoization	189–190, 197, 443
MemoryRouter	392
Meta	27
Metrics	271
Metro bundler	633
Middleware	492
MIT license	27
Mobile devices	631
Mocha	272
Mock	41
Mock Service Worker	300
Module system	82

Monorepo	475
Mounting	169
Multipage application	28
Mutation type	514

N

Named export	82
Native app	631
Navigation	40, 383
Navigation bar	387
Next.js	32, 557, 569
<i>API routes</i>	573
<i>dynamic route</i>	573
<i>getServerSideProps</i>	571
<i>handler function</i>	574
<i>initialization</i>	569
<i>Page component</i>	569
Nginx	384
node_modules	58, 67
Node.js	53
No new features	32
NPM	27, 53, 56, 407
<i>commands</i>	56
npm link	416
npm outdated	59
NPM package	416
NPM registry	418
npm run build	71, 609
npm start	63, 77
nth-child	263
nth-of-type	263

O

Object structure	110
<i>deep</i>	110
Offline capability	605, 618
Omit type	321
Open source	27
OpenSSL	607
Operation, asynchronous	476
Optimization	197

P

Package	
<i>installation</i>	418
<i>publishing</i>	418
package.json	55–56, 66
<i>scripts</i>	63
package-lock.json	67

React (concepts)	34	FlatList component	638
React application		flex layout	642
<i>requirements</i>	53	Form	651
react-codemod	61	Google Play Store	631, 657
React developer tools		input field	644
<i>Components</i>	70	installation	631
<i>Profiler</i>	70	installing the router	651
react-dom	37, 55	interaction option	642
react-error-boundary	597	iOS	631
React Hook Form	305, 334, 369, 449	iOS build	658
<i>error</i>	337	Java	631
<i>error message</i>	342	JavaScriptCore	631
<i>formState</i>	337	keyboard shortcuts	648
<i>handleSubmit</i>	334	keyExtractor prop	639
<i>loading data</i>	336	Kotlin	631
<i>register</i>	334	native GUI elements	636
<i>Resolver</i>	342	NativeRouter	651
<i>validation</i>	337	Objective-C	631
<i>validation rules</i>	339	outer distance	641
<i>validation schema</i>	339	placeholder prop	646
react-i18next	535	project structure	632
<i>browser language</i>	541	publishing	657
<i>dynamic values</i>	544	React Dev Tools	649
<i>installation</i>	535	react-native-elements	652
<i>language switching</i>	542	React Router Native	651
<i>number format</i>	546	renderItem prop	639
<i>placeholders</i>	544	route parameters	652
<i>singular and plural</i>	551	Routes	651
<i>t function</i>	539	separator element	641
<i>useTranslation</i>	536	server communication	646
Reactive Extensions	491	simulator	632, 635
React Native	37, 631	start	633
<i>Android</i>	631	structure	631
<i>Android build</i>	658	style props	643
<i>app.json</i>	657	StyleSheet	640
<i>Apple App Store</i>	631	styling	639
<i>App Store</i>	657	Swift	631
<i>autoCapitalize</i>	646	templates	632
<i>automatic reload</i>	637	Text component	637
<i>Axios</i>	646	TextInput component	644
<i>build</i>	657	TypeScript	638
<i>component library</i>	652	useNavigate	655
<i>controlled component</i>	644, 655	useParams	654
<i>create</i>	640	View component	637
<i>CSS syntax</i>	640	webview	631
<i>debugging</i>	648	worker process	648
<i>device</i>	635	Xcode	631
<i>editing</i>	650	XMLHttpRequest API	646
<i>Emotion</i>	642	React Query	592
<i>error message</i>	637	<i>cache</i>	595
<i>Expo Go app</i>	635	<i>installation</i>	592
<i>Fetch API</i>	646	<i>isError</i>	595

<i>isLoading</i>	595	<i>action creator</i>	458
<i>mutations</i>	595	<i>Action fulfilled</i>	460
<i>Suspense</i>	596, 598	<i>Action pending</i>	460
<i>useMutation</i>	596	<i>Action rejected</i>	460
<i>useQuery</i>	595	<i>action type</i>	444
<i>useQueryClient</i>	596	<i>aggregation</i>	441
React Router	383, 449, 587	<i>AnyAction</i>	479
<i>conditional redirects</i>	396	<i>applyMiddleware</i>	456
<i>dialog handling</i>	400	<i>async middleware</i>	456
<i>dynamic routes</i>	400	<i>configureStore</i>	431
<i>element</i>	386	<i>Context</i>	432
<i>error page</i>	390	<i>createAsyncThunk</i>	459
<i>initialEntries</i>	395	<i>createSelector</i>	443
<i>installation</i>	383	<i>createSlice</i>	436–437
<i>Navigate component</i>	386	<i>creating a data record</i>	448
<i>not found</i>	390	<i>debugging</i>	433
<i>Outlet</i>	402	<i>deleting</i>	465
<i>path</i>	386	<i>design tools</i>	443
<i>priority</i>	387	<i>dispatch</i>	444
<i>Routes component</i>	385–386, 390	<i>dynamic selector</i>	443
<i>subroutes</i>	400	<i>editing a data record</i>	448
<i>testing</i>	392	<i>error handling</i>	464, 470
<i>useLocation</i>	402	<i>generators</i>	473
<i>useNavigate</i>	399, 405	<i>higher-order selectors</i>	442
<i>useParams</i>	405	<i>initialState</i>	437
<i>variable</i>	387	<i>installation</i>	431
<i>variables</i>	400	<i>loading data</i>	463
React Scripts	62	<i>loadingState</i>	462, 467
<i>build</i>	62	<i>middleware</i>	433, 455, 475
<i>eject</i>	63	<i>overhead</i>	431
<i>start</i>	62	<i>PayloadAction</i>	446
<i>test</i>	63	<i>reducer</i>	435
react-scripts	55	<i>rejectWithValue</i>	460
<i>test</i>	273	<i>Reselect</i>	443
React Testing Library	294, 296	<i>RootState</i>	432
<i>fireEvent</i>	298	<i>selector cache</i>	444
<i>getAllByTestId</i>	297	<i>selector function</i>	442
<i>getByTestId</i>	297	<i>selectors</i>	440, 443
react-test-renderer	289	<i>server communication</i>	435
Readability	197	<i>side effects</i>	435, 455, 473
Readme	66	<i>slice</i>	435
Reconciler	28, 30, 37	<i>store</i>	431
<i>algorithm</i>	38	<i>store access</i>	438
<i>assumptions</i>	38	<i>strict workflow</i>	427
<i>optimization</i>	38	<i>structure</i>	431
Reducer	40, 430	<i>substate</i>	435
Reducer function	192	<i>typesafe-actions</i>	476
Reducer hook	190	<i>useAppDispatch</i>	445
<i>asynchronicity</i>	193	<i>useSelector</i>	439
<i>middleware</i>	193	Redux Dev Tools	433, 455
Redux	40, 213, 427	Redux middleware	
<i>action</i>	444	<i>action</i>	456

-
- next* 456
 - store* 456
 - Redux Observable 491
 - action\$* 494
 - combineEpics* 494
 - creating* 496
 - deleting* 495
 - Epic middleware* 492
 - installation* 492
 - modifying* 496
 - ofType* 494
 - operator chain* 492
 - pipe* 494
 - read access* 493
 - RootEpic* 494
 - running* 492
 - switchMap* 494
 - TypeScript* 492
 - Redux Saga 473
 - action creator* 476
 - all* 478
 - creating* 487
 - deleting* 482
 - error handling* 477
 - fork* 478
 - installation* 475
 - loading data* 476
 - Middleware* 475
 - modifying* 487
 - rootSaga* 476
 - saga* 476
 - takeLatest* 477
 - TypeScript* 475
 - Redux Thunk 455, 457
 - creating* 470
 - integration* 457
 - modifying* 470
 - reading data* 459
 - server communication* 458
 - Redux Toolkit 431
 - Ref 189, 305–306
 - current* 308–309
 - Reference 306
 - Referencing 91
 - Ref hook
 - form* 200
 - storing values* 201
 - Relay 515, 592
 - Release 28
 - rename-unsafe-lifecycles 179
 - Render 34, 170
 - Render as you fetch 599
 - Renderer 27, 37
 - Rendering 74
 - Rendering process 106
 - render method 172, 577
 - Render priority 207
 - Render process 116
 - Render prop 146
 - integration* 149
 - naming* 147
 - reportWebVitals 75–76
 - Cumulative Layout Shift* 76
 - First Contentful Paint* 77
 - First Input Delay* 76
 - Largest Contentful Paint* 76
 - Time to First Byte* 77
 - Request for comments (RFC) 29
 - Requirements
 - browser* 53
 - editor* 53
 - Node.js* 53
 - Re-render 166
 - Resources 123, 555
 - REST 509
 - Reuse 407
 - Roboto 347
 - Rollup 408
 - Root component 74, 77
 - Root element 86
 - Root node 32
 - Root object 48
 - Router 40
 - Rules, file 80
 - Rules of Hooks 215
 - RxJS 491
 - observable* 492
 - observer* 492
 - operator* 492
- ## S
-
- Safari 53
 - Saga 474
 - Sass 252
 - mixins* 252
 - nesting* 252
 - variables* 252
 - Scrolling, virtual 601
 - SCSS 252, 342
 - @import* 253
 - color variables* 252
 - Search engine 555
 - Security 51

Security mechanism	65	<i>.storybook directory</i>	423
Selective updates	121	<i>component story</i>	423
Semantic versioning	29, 33, 187	<i>initialization</i>	422
<i>major version</i>	29	StrictMode	75–76, 121, 464
<i>minor version</i>	29	Structure	66
<i>patch level</i>	29	<i>components</i>	68
Server communication	64, 126	<i>directories</i>	67
<i>error</i>	129	<i>files</i>	66
<i>error handling</i>	130	<i>style definitions</i>	68
<i>exception</i>	130	<i>tidying up</i>	74
Server-side rendering	28, 212, 555	Styled Components	257
shouldComponentUpdate	174, 580	Styled components	
Side effect	116, 119, 173	&&&	264
Single-page application (SPA)	28, 383, 555	<i>dynamic styling</i>	263
Smart component	137	<i>extension</i>	265
Snowpack	61	<i>file</i>	261
SOAP	509	<i>pseudo selectors</i>	262
Source code		<i>styled function</i>	265
<i>organization</i>	306	Style guide	78
src directory	68	style prop	253
SSL certificate	607	Stylesheet, global	246
<i>subject alternative name</i>	608	Styling	245
<i>trust</i>	608	submit event	310–311
SSR	555	super	168
<i>Babel configuration</i>	560	Suspense	31, 587
<i>dynamics</i>	565	Suspense for Data Fetching	31, 591
<i>functionality</i>	556	SVG	362
<i>hydrateRoot</i>	556–557	Switching pages	383
<i>hydration</i>	564	Symbolic link	416
<i>implementation</i>	557	Syntax highlighting	246
<i>initialization</i>	557	Synthetic event	328
<i>JSX code</i>	558	<i>wrapper</i>	328
<i>NPM scripts</i>	562		
<i>render</i>	557	T	
<i>renderToString</i>	562		
<i>server communication</i>	557	Table, virtual	599
<i>Webpack</i>	558	Tagging function	261
<i>Webpack configuration</i>	560	Tailwind	266
Standard configuration	55	<i>@tailwind</i>	266
startTransition	210	<i>default classes</i>	267
State	36, 42, 100, 427	<i>responsiveness</i>	268
<i>initial value</i>	101	<i>theme</i>	268
<i>rendering</i>	101	TDD	
State management	40, 427	<i>Green</i>	275
<i>changes</i>	429	<i>Red</i>	275
<i>data store</i>	428	<i>Refactor</i>	275
State object	107	Template string	258
Static asset	67	Terser	584
Static type checking	219	Test	271
Status management		<i>automated</i>	272
<i>central</i>	455	<i>configuration issues</i>	271
Storybook	422	<i>documentation</i>	271

U

useEffect 116, 188
 async/await 131
 dependencies 116, 118, 121
 dependency 122
useId 189, 212
useImperativeHandle 189, 202
useInsertionEffect 189, 213
useLayoutEffect 189, 205
useMemo 189, 580
use prefix 214
useReducer 189–190
useRef 124, 189, 306
User interface 33
useState 100, 188
 return value 102
useSyncExternalStore 189, 213
useTransition 189, 210
 in process 212
Utility-first framework 266

V

V8 53
Validation 312
 error 315
 forms 334
 server-side 312
 test 315
 testing 343
var 232
Verdaccio 418
View layer 27
Visual Studio Code 53

Vite 61
Vue.js 28, 49

W

Walke, Jordan 29
Webpack 54, 245, 408, 584, 633
 configuration 63
 dev server 55, 77
Web server 51
WebSockets 123
WebStorm 53
Workbox 618
Workbox CLI 621

X

XE “Concurrent renderer” 32
XE “TypeScript
 class” 233
XHP 29
XSS injections 537
XSS protection 89

Y

Yarn 57
 caching 57
 integrity 58
 plug and play 59
 security 58
yield 474
Yup 339

Service Pages

The following sections contain notes on how you can contact us.

Praise and Criticism

We hope that you enjoyed reading this book. If it met your expectations, please do recommend it. If you think there is room for improvement, please get in touch with the editor of the book: rachelg@rheinwerk-publishing.com. We welcome every suggestion for improvement but, of course, also any praise!

You can also share your reading experience via Twitter, Facebook, or email.

Supplements

If there are supplements available (sample code, exercise materials, lists, and so on), they will be provided in your online library and on the web catalog page for this book. You can directly navigate to this page using the following link: <http://www.rheinwerk-computing.com/5556> Should we learn about typos that alter the meaning or content errors, we will provide a list with corrections there, too.

Technical Issues

If you experience technical issues with your e-book or e-book account at Rheinwerk Computing, please feel free to contact our reader service: support@rheinwerk-publishing.com.

About Us and Our Program

The website <http://www.rheinwerk-computing.com> provides detailed and first-hand information on our current publishing program. Here, you can also easily order all of our books and e-books. Information on Rheinwerk Publishing Inc. and additional contact options can also be found at <http://www.rheinwerk-computing.com>.

Legal Notes

This section contains the detailed and legally binding usage conditions for this e-book.

Copyright Note

This publication is protected by copyright in its entirety. All usage and exploitation rights are reserved by the author and Rheinwerk Publishing; in particular the right of reproduction and the right of distribution, be it in printed or electronic form.

© 2024 by Rheinwerk Publishing, Inc., Boston (MA)

Your Rights as a User

You are entitled to use this e-book for personal purposes only. In particular, you may print the e-book for personal use or copy it as long as you store this copy on a device that is solely and personally used by yourself. You are not entitled to any other usage or exploitation.

In particular, it is not permitted to forward electronic or printed copies to third parties. Furthermore, it is not permitted to distribute the e-book on the Internet, in intranets, or in any other way or make it available to third parties. Any public exhibition, other publication, or any reproduction of the e-book beyond personal use are expressly prohibited. The aforementioned does not only apply to the e-book in its entirety but also to parts thereof (e.g., charts, pictures, tables, sections of text).

Copyright notes, brands, and other legal reservations as well as the digital watermark may not be removed from the e-book.

Digital Watermark

This e-book copy contains a **digital watermark**, a signature that indicates which person may use this copy. If you, dear reader, are not this person, you are violating the copyright. So please refrain from using this e-book and inform us about this violation. A brief email to info@rheinwerk-publishing.com is sufficient. Thank you!

Trademarks

The common names, trade names, descriptions of goods, and so on used in this publication may be trademarks without special identification and subject to legal regulations as such.

All products mentioned in this book are registered or unregistered trademarks of their respective companies.

Limitation of Liability

Regardless of the care that has been taken in creating texts, figures, and programs, neither the publisher nor the author, editor, or translator assume any legal responsibility or any liability for possible errors and their consequences.